MOTION DESCRIPTION LANGUAGES: FROM SPECIFICATION TO EXECUTION

A Thesis
Presented to
The Academic Faculty

by

Patrick J. Martin

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2010

MOTION DESCRIPTION LANGUAGES: FROM SPECIFICATION TO EXECUTION

Approved by:

Professor Magnus Egerstedt, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Linda Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Jeff Shamma
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Michael Stilman
College of Computing
Georgia Institute of Technology

Professor Ayanna Howard
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 19 March 2010

To Pop: you taught me to never give up on my dreams.

To Mom and Dad: you enabled the pursuit of my dreams.

To Julie: your love makes my dreams turn into reality.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Magnus Egerstedt, for letting me take part in interesting and challenging research problems during my Ph.D. program at Georgia Tech. His constant enthusiasm and support helped make my research ideas come to fruition. I would also like to thank the faculty members who served on my dissertation committee, Jeff Shamma, Ayanna Howard, Linda Wills and Micahel Stillman, for their time and effort.

Additionally, I would like to thank the many professors who inspired me over the years. In particular, Lee Cohen, Mike McDermott, and Stan Cheyne at Hampden-Sydney College, as well as P.S. Krishnaprasad at the University of Maryland. Without their encouragement, I would not have returned to graduate school.

While pursuing my research, I had the pleasure of collaborating and working with several outstanding people. I want to thank Todd Murphey and Elliot Johnson at Northwestern University for their helpful collaboration on our robotic puppetry work. Also, I want to thank the members of the GRITS Lab for their many interesting and helpful discussions: Amir Rahmani, Brian Smith, Dennis Ding, Musad Haque, Peter Kingston, Rahul Chipalkatty, JP de la Croix, Jonghoek Kim, and Philip Twu.

Finally, I owe a debt of gratitude to my family, who gave me unconditional love and support over the years. I want to thank my parents and brother for their encouragement during my academic career. And last, but not least, I would like to thank my wife, Julie, for always being there throughout the highs and lows over the past few years of school! We made it!

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

Many emerging controls applications have seen increased operational complexity due to the deployment of embedded, networked systems that must interact with the physical environment. In order to manage this complexity, we design different control modes for each system and use motion description languages (MDL) to specify a sequence of these controllers to execute at run-time. Unfortunately, current MDL frameworks lose some of the important details (i.e. power, spatial, or communication capabilities) that affect the execution of the control modes. This work presents several computational tools that work towards closing MDL's specification-to-execution gap, which can result in undesirable behavior of complex systems at run-time. First, we develop the notion of an MDL compiler for control specifications with spatial, energy, and temporal constraints. We define a new MDL for networked systems and develop an algorithm that automatically generates a supervisor to prevent incorrect execution of the multi-agent MDL program. Additionally, we derive conditions for checking if an MDL program satisfies actuator constraints and develop an algorithm to insert new control modes that maintain actuator bounds during the execution of the MDL program. Finally, we design and implement a software architecture that facilitates the development of control applications for systems with power, actuator, sensing, and communication constraints.

CHAPTER I

INTRODUCTION AND BACKGROUND

1.1 Introduction

Many emerging controls applications have seen increased system complexity due to the pervasive use of computing and networking resources [54]. In particular, recent advances in wireless technology and embedded computing have enabled the proliferation of systems that interact with the physical world. The so-called *cyber-physical* systems (CPS) can also be deployed to instrument and control large scale systems, from buildings to sensor networks. In order to manage this complexity, system abstractions and specification languages are designed so that control tasks can be broken up into smaller "chunks," or symbols, which are then interpreted by the systems at run-time.

This abstraction-based approach to control design lets users specify a desired system execution using pre-defined control modes. However, current system abstractions lose some of the important details that affect the *execution* of the control modes. This *specification-to-execution* gap can result in undesirable behavior at run-time. It would be beneficial if there existed an intermediate process, such as the one illustrated in Figure 1, that takes a system specification and brings it in-line with system capabilities and constraints before execution.

One challenge for this compilation process is the inherent heterogeneity of CPS due to their varying computational capabilities. Additionally, actuators and sensors may be different among types of a single class of systems, such as a collection of mobile robots. These robots may have the same drive train and the physical dynamics; however, each robot may be equipped with different suites of sensors from IR to

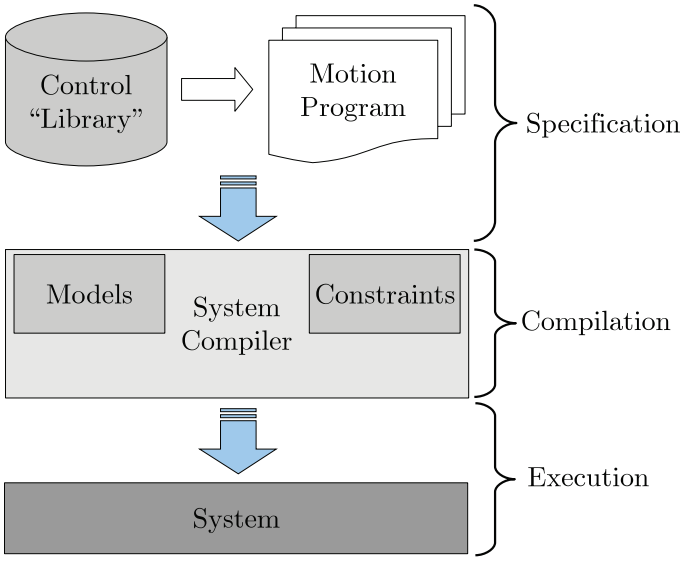


Figure 1: This work focuses on developing the tools necessary for the development of an intermediate, or "compilation," layer for creating motion programs for cyber-physical systems.

GPS. The selection of sensors affects the ability of the robot to execute control tasks. Consequently, new specification languages are required to enable the specification of control tasks among systems with different capabilities.

Once we have properly formulated these specification languages, we need to coordinate their requirements with the system capabilities. This task is important because the controllers encoded within these symbolic specifications have to operate in an environment with real constraints, such as actuator limits or power consumption. To handle these challenges, new frameworks for symbolic control must be developed that allow for the "compilation" of control symbols into executable control code for the target system.

The purpose of this research is to develop design and optimization tools for the generation of executable control code from high-level, symbolic specifications. In particular, this thesis addresses the following challenges:

- *Spatio-temporal optimization of motion programs*: How do we ensure that motion programs for systems with time and space constraints execute correctly?

- *Multi-agent motion program specification*: How do we specify motion programs for multi-agent systems? How do we prevent our specification from causing incorrect execution when deployed on our target platforms?

- *Expanding motion programs under input constraints*: When our system has hard constraints on actuators, how do we adjust our motion program to execute without violating the bounds?

- *Enabling tools for cyber-physical system development*: When we have specified control for complex systems, what software is necessary to enable the on-line adjustment of control algorithms and system capabilities?

The organization of this thesis is as follows: the remainder of this chapter introduces the background research and theory that provided a starting point for this research. Chapter 2 presents our work on developing motion programs and optimizing their execution for systems with spatial and temporal constraints. The work in Chapter 3 addresses the problem of specifying motion programs for distributed systems and automatically generating statically consistent multi-agent programs. In Chapter 4, we develop an algorithm for modifying motion programs when the system has input constraints. Chapter 5 develops and demonstrates our software architecture to enable the dynamic adjustment of controllers and system capabilities on CPS platforms. Finally, we conclude with a summary and final remarks in Chapter 6.

1.2 Background Research

Since cyber-physical systems interact with the real world, they are designed to react and possibly change their mode of operation while deployed. In order to develop theoretical and computational tools for generating control code for these systems,

we need to understand some concepts from hybrid system theory and discrete event systems. The high-level *specification* for controlling these hybrid systems is also important; consequently, we discuss the basic MDL formalism to be used in the proposed research. Finally, we describe the recent methods for switched-time optimization of hybrid systems, which we see as a fundamental part of the control code generation task.

1.2.1 Hybrid Systems

Most modern control systems use computers and digital sensors to interface and control systems within continuous environments. The field of *hybrid system theory* encompasses the analysis and design of these complex embedded systems. A common example of a hybrid system is the thermostatically controlled room [12]. The room temperature is modeled using continuous thermal dynamics; however, the input of heat into the room is controlled by one of several distinct controllers, or *states*. The system then evolves by moving through a finite state machine, composed of the operational states of the controllers.

Although hybrid systems have been examined as far back as [58], more recent discussions of the modeling, analysis, and control of hybrid systems are seen in [11, 28]. In its most general form, a hybrid system is composed of a collection of systems of differential equations, equipped with maps for jumping between them. These jumps may be *autonomous*, such as when the state of the system hits a particular region of the state space. Additionally, the jumps may be *controlled*, which allows a choice of how to jump and where to jump in the state space. We can write the basic model of a *controlled hybrid system* as [12]:

$$H = (Q, \mathbf{\Sigma}, \mathbf{A}, \mathbf{G}, \mathbf{C}, \mathbf{F}). \tag{1}$$

In this equation, Q represents the finite states of the hybrid system and $\mathbf{\Sigma}$ is a collection of differential equations such that each $q \in Q$ has its own dynamic equations, Σ_q. In other words, each discrete state, q, has a differential equation

$$\dot{x}_q = f_q(x_q, u_q)$$

that maps the state space, X_q, and input space, U_q, to some real vector space $\mathbb{R}^n$. Additionally, the hybrid system has a combined *hybrid system state space*, which is defined by the product $S_q = X_q \times \{q\}$ [12].

The set $\mathbf{A}$ denotes the collection of *autonomous jump sets*, or sets which cause a jump to another discrete state via an associated map in $\mathbf{G}$. In other words, if the state of the system enters some jump set $A_i \in \mathbf{A}$ (where i is some index from $\mathbb{N}$), a map $G_i \in \mathbf{G}$ will transition the system to another state. The *controlled jump sets*, denoted by $\mathbf{C}$, denote the sets where a jump may or may not occur. These jumps are provided via the collection of *controlled destination maps*, $\mathbf{F}$, which take points in some controlled jump set $C_i \in \mathbf{C}$ and maps them to some other state of the hybrid system. For example, if the state has entered the set C_i, a controller has the option to apply the map F_i, which transfers the system state to another hybrid state, S_q.

1.2.1.1 Hybrid Automata

The preceding definitions are quite general and are best digested by using *hybrid automata* to visually represent the model of the hybrid system. These automata apply the finite automata model to the definition from (1), which enables us to more readily see how the system can evolve during execution. As an example, consider the hybrid automaton of Figure 2. This automaton has three states, $Q = \{q_1, q_2, q_3\}$, and each of the states' dynamics are listed within the state, such as q_1's linear system

$$\Sigma_{q_1} = \begin{cases} \dot{x} = A_1 x + B_1 u \\ y = C_1 x, \end{cases}$$

where $x \in \mathbb{R}^n, u \in \mathbb{R}^m,$ and $y \in \mathbb{R}^p$. Note that the q_1 state may transition to *either* q_2 or q_3, depending on the values of $||x||$ and $||u||$. The edge labels are logical statements, representing the possible autonomous jump sets ($A_i \in \mathbf{A}$) for the system as it evolves over time. Additionally, there is a reset condition specified in the diagram, denoted $x := x_0$, which represents a jump map, $G_{q_3}(x) = x_0$, that changes the state to its initial condition whenever a transition from q_2 to q_3 occurs.

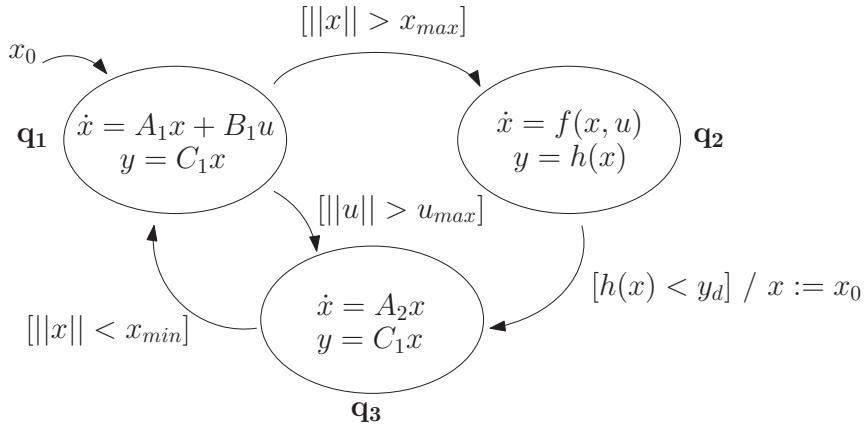


Figure 2: A hybrid system automaton with three states, $q_1, q_2,$ and q_3. In this case, q_1 and q_3 have linear dynamics and q_2 has non-linear dynamics.

1.2.1.2 Finite Automata and Languages

The prior description of hybrid automata was inspired by the finite automata model, as described in [17, 51]. A finite automaton is defined by the tuple

$$G = (Q, E, f, \Gamma, q_0, Q_m).$$

The set Q represents the states of the automaton and E is the set of events, or transitions, among the states in G. These events allow the definition of the mapping $f : Q \times E \to Q$ that provides the dynamics of the evolution of the automaton, i.e. $f(q_1, e_1) = q_2$ means that the event e_1 allows the transition from q_1 to q_2. The function $\Gamma : Q \to 2^E$ determines what events are possible at some state in Q. Finally, q_0 is

the initial state and Q_m is a set of *marked* states that are used to identify particular states in G.

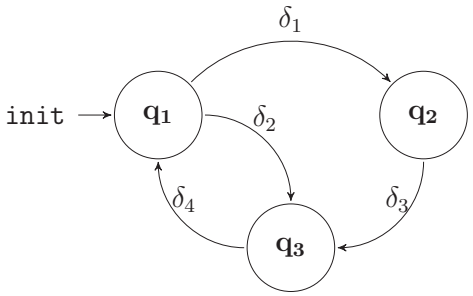


Figure 3: The hybrid automaton from Figure 2 abstracted as the finite automata, H.

These models are useful when a system switches state due to some transition, or event, occurring. For example, we could abstract the example from Figure 2 and only consider the states and transitions from a high-level, as shown in Figure 3. In this figure, each state from the hybrid automaton is labeled by its state ($q_1, q_2,$ and, q_3) and each edge is assigned a label, δ, that represents the transition from one state to another. When this finite automaton executes, we know what state it is currently in by examining the string of transition labels. In other words, if the string $\delta_1\delta_3\delta_4$ is processed, we know that the automata is currently in state q_1.

These automata generate a *language* based on the transitioning events among the states. The transition labels in Figure 3 are considered part of a finite set, or *alphabet*, $\mathcal{A} = \{\delta_1, \delta_2, \delta_3, \delta_4\}$. By concatenating symbols from this set, we construct strings that represent a sequence of events that have occurred. We make use of the empty string, ϵ, to represent that no event has occurred. Consequently, the set of *all* concatenations of symbols from $\mathcal{A}$ is represented by $\mathcal{A}^\star$. Since the automaton in Figure 3 has a structure imposed by the transitions, it generates a particular language, $L(H) \subseteq \mathcal{A}^\star$. In other words, the sequence $\delta_1\delta_3\delta_4$ is accepted by the automaton; however, $\delta_1\delta_2\delta_4$ is not accepted.

1.2.2 Motion Description Languages

The work in [13] uses the prior concepts from language theory and hybrid systems to solve problems involving the computer control of motion. The author proposes a motion programming language, called MDL, for decomposing large control problems into its smaller components. The inspiration for this novel work stems from the increased use of computer controlled machines in factory settings, such as robot manipulators and automated machining tools [14] as well as the fundamentals of computer programming languages [29]. The primary advantages of MDL are the reduced complexity for control task specification and device independence of the controllers, which enables broader application of the control tasks among different systems.

We define an MDL as a set of strings made up of symbols that encode control laws and their execution times. The system accepts these symbols sequentially, executing the control law and transitioning between elements at specified times. Specifically, if we model a MDL device as the differential equation

$$\begin{aligned} &\dot{x}(t) = f(x(t), u(t)),\ x \in \mathcal{X} \subseteq \mathbb{R}^n,\ u \in \mathcal{U} \subseteq \mathbb{R}^m \\ &y(t) = h(x(t)),\ y \in \mathcal{Y} \subseteq \mathbb{R}^p. \end{aligned} \tag{2}$$

then the input of symbols, or MDL modes, force this system along a trajectory, $x(t)$, that approximates the solution to (2) [13]. The control signal, $u(t)$, is generated according to the current MDL mode's encapsulated feedback (or, open-loop) control law, which we define as a mapping $\kappa : \mathcal{Y} \to \mathcal{U}$. We construct MDL modes, denoted σ, by pairing a control law, κ, with a timer, τ, such that $\sigma = (\kappa, \tau)$.

For example, if we are given a string of M MDL symbols,

$$(\kappa_1, \tau_1)(\kappa_2, \tau_2) \cdots (\kappa_M, \tau_M)$$

then the system (2) evolves according to

$$\begin{aligned} \dot{x} &= f(x, \kappa_1),\ t \in [t_0, \tau_1) \\ \dot{x} &= f(x, \kappa_2),\ t \in [\tau_1, \tau_2) \\ &\vdots \\ \dot{x} &= f(x, \kappa_M),\ t \in [\tau_{M-1}, \tau_M], \end{aligned} \tag{3}$$

where t_0 is some intial time, not necessarily zero. In other words, the system starts with the mode (κ_1, τ_1), applying the control signal $u(t) = \kappa_1(y(t))$ for τ_1 seconds. At the end of that time slice the second mode is loaded and a new control signal, $u(t) = \kappa_2(y(t))$, pushes the system until its timer fires. The process continues until there are no remaining symbols to feed into the device.

The authors of [40, 39, 30] generalize the switching conditions to include sensor feedback, so systems can react to dynamic environment changes and still use MDL motion programs. These *interrupt functions*, $\xi : \mathcal{Y} \to \{0, 1\}$, augment the MDL mode by creating a triple: $\sigma = (\kappa, \xi, T)$. This extended MDL, or *MDLe*, is successfully used in mobile robot systems, where operation in dynamic environments requires a more flexible control switching framework.

Note that the inclusion of an interrupt function affects the evolution of the system by changing the timing bounds of each modes execution. Instead of always switching at the set time τ, the interrupt function may fire beforehand at some time $\tau_\xi \leqslant \tau$. Consequently, the execution times of the modes in (3) become $t \in [\min[\tau_{\xi_{i-1}}, \tau_{i-1}], \min[\tau_{\xi_i}, \tau_i]]$ for $i = 1, \cdots, M$. Furthermore, the work in [30], reformulates the interrupt function into a combined mapping $\xi : \mathcal{Y} \times \mathbb{R}^+ \to \{0, 1\}$. Applying this timer absorption, we state the definition of an MDL mode:

Definition 1.1 (MDL Mode)**.** A MDL mode is a tuple (κ, ξ), composed of a control law, $\kappa : \mathcal{Y} \to U$, and a transition function, $\xi : \mathcal{Y} \times \mathbb{R}^+ \to \{0, 1\}$.

We can visualize an MDL based system with a hybrid automaton, as described

in Section 1.2.1.1. In Figure 4 a sequence of modes, or the system states, are shown. MDL systems do not use reset conditions to move the state of the system to some other value. Instead, the system dynamics are altered by selecting an input signal, $u_i(t) = \kappa_i(y(t))$ for $i = 1, \cdots, M$. For example, this system is initially started at some $x(0) = x_0$ with the controller $u_1(t) = \kappa_1(y(t))$. As it executes the dynamics with this controller, the current transition function ($\xi_1(y(t))$) "monitors" the output and the system timer is advances. If *either* the $\xi_1 \rightarrow 1$ *or* $t \geq \tau_1$, the system will change its controller to the next one in the sequence: κ_2. This process repeats until the final interrupt occurs or terminal time is reached, resulting in the final state of the system, x_f.

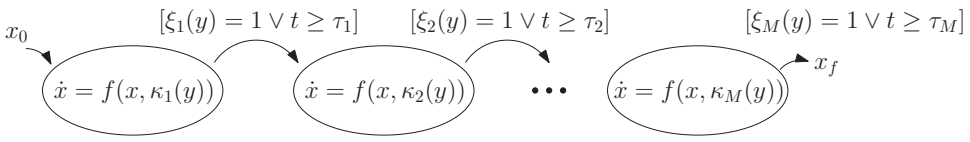


Figure 4: The hybrid automaton of the system evolution in equation (3) with the incorporation of MDLe's interrupt function, $\xi(\cdot)$. Note that the MDL based hybrid system is sequential in nature with only *one* transition between states.

1.2.2.1 Recent Work in MDL

Several recent research papers analyze MDL and its extension, MDLe. In [22, 21] the *specification complexity* of a MDL program is introduced. This complexity measure determines the number of bits necessary to accomplish a control task with a set collection of controller. Additionally, the work designs a feedback mechanism by defining a *free-running, feedback automata*, which is a specialized automata used to reduce the specification complexity of the program. Follow up work in specification complexity is seen in [20] where minimal strings of MDL modes are generated from streams of input/output data.

Some additional research works towards enabling multi-agent systems to use MDLe motion programs as a specification language for group behavior. Part of the work in

[64] introduced the concept of a *group atom*, which is an atom with global control and interrupt functions. Although group atoms do allow for a set of agents to perform an action together, it forces a centralized system on the agents. Alternatively, [65] proposes a process for composing separate MDLe strings so that interaction between MDLe based systems can be analyzed. The authors successfully illustrate their MDLe composition process with the simulation of a pair of heterogeneous agents.

1.2.2.2 MDL as Symbolic Control

This research uses MDL as the high-level specification language for scripting motion programs. However, MDL is just one example of a specification language for hybrid systems, and other types of symbolic control are discussed here to illustrate that control tasks can be abstracted in different ways. According to the authors of [7], there are two approaches to discretizing the control task: control-driven and environment-driven. Control-driven frameworks are useful when the dynamics of the system are complicated or when the environment changes due to moving obstacles or boundaries. Alternatively, environment-driven frameworks use geometry within its specification to break up the control task.

Maneuver automata [25, 26] use a control-driven approach for discretizing the control task. In particular, maneuver automata use a set of input symbols known as *motion primitives*, which are specialized forms of MDL modes that operate on systems satisfying the property of invariance under group actions. The simplest primitives, known as *trim* primitives, are symbols which drive the system with constant control actions. Alternatively, the *maneuver* primitives perform more complicated trajectories; however, they must begin and end at the system's steady-state conditions. The authors use maneuver automata to plan motion for unmanned helicopters using a finite collection of pre-defined motion primitives. The resulting plans are generated by inserting available maneuvers between trim primitives such that particular feasibility

conditions are satisfied. An advantage of this approach is that the final state for a particular plan may be derived without resorting to numerical integration. However, this approach limits the types of systems that maneuver automata can drive.

Another approach that uses control-driven discretization is *feedback encoding* using control quanta [8, 9]. These control quanta are identical in definition to the MDL modes defined in [13], i.e. each symbol is defined as a tuple of control action and time interval, (u, τ). Since the control quanta use *only* time-based transitions, additional algebraic structure can be imposed on these symbols. This structure enables the authors to generate control quanta plans with some notion of plan "efficiency." Unfortunately, control quanta are not well suited in dynamic environments since their execution is entirely based on time.

Unlike the previous task discretization methods, *linear temporal logics* take advantage of discretizing the environment in which hybrid systems operate. Some examples of recent work in this field are [32, 56] with extensions and applications for multi-agent systems detailed in [33]. LTL use symbols that model objects or regions of interest in an environment and then apply Boolean logic operators in order to form logic sentences. This approach is highly structured over a static environment; consequently, the LTL sentences can be analyzed with a model checking process. The final plan generated by the model checking process is provably correct. One disadvantage of the LTL approach is the requirement to have a static environment. Obstacles and other features must be known in advance in order to prevent incorrect behaviors. Furthermore, the authors of [7] mention that LTL specifications are unable to control more realistic systems with non-holonomic constraints.

1.2.3 Optimal Control of Switched Systems

MDL specify the motion control of a hybrid system using a string of control laws, and, as seen in Section 1.2.2, they cause the system to *switch* current mode at particular

times. What if these switch times violate our desired timing constraints? One way to answer this question is to formulate an optimal control problem with the switch times of the hybrid system acting as the control parameters. In other words, given a system of the form

$$\dot{x} = f_i(x(t), u(t), t),\ t \in [\tau_{i-1}, \tau_i],\ i = 1, \cdots, M, \tag{4}$$

minimize the cost functional

$$J(\overline{\tau}) = \int_{\tau_0}^{\tau_M} L(x(t), u(t))dt. \tag{5}$$

with respect to the control parameters $\overline{\tau} = [\tau_1 \cdots \tau_{M-1}]$.

The recent work in [55] characterizes the *hybrid maximum principle* for problems of this type. The author defines the *costate* of the system (4) as $\lambda(t)$ and its *backward* differential equation dynamics are derived from (4) and (5) as

$$\dot{\lambda}(t) = -\Big(\frac{\partial f_i}{\partial x}(x(t), u(t))\Big)^T \lambda - \Big(\frac{\partial L}{\partial x}(x(t), u(t))\Big)^T,\ t \in [\tau_i, \tau_{i-1}],\ i = M, \cdots, 1.$$

Additionally, the Hamiltonian for this problem is formulated as

$$H(x(t), \lambda(t), u(t)) = \lambda(t) f_i(x(t), u(t)) + L(x(t), u(t)),\ t \in [\tau_i, \tau_{i-1}],\ i = M, \cdots, 1, \tag{6}$$

and the author shows that the optimal switching schedule (i.e. the switch times) and the optimal input trajectory minimize (6).

Following this work, several authors have solved classes of optimal control problems for switched systems and a representative sampling of results are seen in [24, 53, 60, 61]. The approach taken in our research will follow the work of [24], where calculus of variations is used to derive the necessary optimality conditions for the switching times of a hybrid system (4) and a gradient descent algorithm is designed for numerically solving the problem. The optimal control problem in [24] is slightly modified by removing the explicit dependence on $u(t)$ in (4) and (5), resulting in the

new problem:

$$\begin{aligned} \min_{\overline{\tau}} \quad & J(\overline{\tau}) = \textstyle\int_{\tau_0}^{\tau_M} L(x(t),t)\ dt \\ \text{such that} \quad & \dot{x} = f_i(x(t)),\ t \in [\tau_{i-1}, \tau_i],\ i = 1, \cdots, M. \end{aligned} \tag{7}$$

Applying variations to each τ_i, the optimality conditions for this problem are given by

$$\frac{\partial J}{\partial \tau_i} = \lambda^T(\tau_i)\Big(f_i(x(\tau_i)) - f_{i+1}(x(\tau_i))\Big) = 0,\ i = 1, \cdots, M-1 \tag{8}$$

where the costate $\lambda(t)$ is calculated from the modified costate equation

$$\dot{\lambda}(t) = -\Big(\frac{\partial f_i}{\partial x}(x(t),t)\Big)^T \lambda - \Big(\frac{\partial L}{\partial x}(x(t),t)\Big)^T,\ t \in [\tau_i, \tau_{i-1}],\ i = M, \cdots, 1.$$

The gradient descent algorithm computes the directional derivatives for each τ_i, as seen in (8), at each algorithm step, k. Then, if (8) is not within some error value, $\rho > 0$, the algorithm changes each τ_i according to the step

$$\tau_i(k+1) = \tau_i(k) - \gamma \frac{\partial J}{\partial \tau_i},$$

where $\gamma > 0$ is a step size. Once the algorithm converges, the output is a *local* optimal value for $\overline{\tau} = [\tau_1\ \cdots\ \tau_{M-1}]$.

CHAPTER II

SPATIO-TEMPORAL MOTION PROGRAMS

This chapter presents our work on compiling motion programs for systems with timing, energy, and spatial constraints. This research uses a modified form of the original MDL formalism [13]. This new framework facilitates the creation of motion programs that support novel spatio-temporal motion constraints as well as energy parameterized motions. In particular, this work is applied to the problem of robotic puppetry. Puppeteers script plays that designate a string of motions for each character within a structured environment; consequently, the use of MDL strings for *specifying* plays is a natural choice, as observed in [23]. As such we script plays using MDL and take the resulting nominal symbolic descriptions of the play and generate optimized, executable programs based on the system dynamics and an associated cost criterion.

The resulting optimization problem is not unique to puppetry, since MDL-based abstractions of hybrid systems may need to optimize their motion programs in order to account for system dynamics and constraints in a number of other applications. We approach the solution to this problem by drawing from recent results in switched-time optimization [5, 24, 53, 61], focusing on the scheduling of discrete transitions in a hybrid system by adjusting the timing and energy parameters of the program.

We discuss the development of our spatio-temporal MDL and its application to autonomous marionettes in Section 2.1. Then, in Section 2.2, we formulate an optimal control problem that characterizes the typical usage of MDL motion programs and derive optimality conditions that serve as our MDL compiler. Finally, Section 5.3 demonstrates this compiler on a collection of simulated autonomous marionettes as well as a robotic marionette system.

2.1 Spatio-temporal Motion Description Languages

In order to script a motion program we formulate a special MDL that accounts for four important properties of our desired motion programs: *who* should act, *what* motion should they do, *where* should they operate, and *when* should the action occur. We assume that the agents are identified by an index, $i \in \mathcal{M}$, where $\mathcal{M} = \{1, \cdots, m\}$, and each agent has the dynamics,

$$\dot{x}^i = f(x^i, u^i),\ x^i \in \mathbb{R}^n,\ u^i \in \mathbb{R}^p, \tag{9}$$

where we use the superscript i to denote agent-i.

We define the input to this model as one in a collection of possible feedback laws, i.e. $u^i = \kappa_j(x^i, t, \alpha_j)$, with κ_j, for some j, coming from a finite set of control laws $\mathcal{K} = \{\kappa_1, \cdots, \kappa_C\}$; additionally, α_j is an "energy"-scaling parameter that could affect speed, amplitude, or some other property of the control mode. When applying a controller of this form, we get the resulting closed-loop system dynamics $\dot{x}^i = f(x^i, t, \kappa_j(x^i, t, \alpha_j))$.

We combine controllers from $\mathcal{K}$ with a time-driven interrupt, denoted τ, that dictates the time at which the control mode interrupts, resulting in controller-interrupt pairs of the form (κ, τ). However, to allow for the specification of programs involving multiple agents, we add in an element for agent identification, i, and a spatially defined location, r, where the agent performs its control κ. These locations in the environment come from a set $\mathcal{R} = \{r_1, \cdots, r_l\}$. Using these additional elements, we thus define our MDL mode below:

Definition 2.1 (Spatio-temporal MDL Mode)**.** An spatio-temporal MDL mode is the tuple $(i, \kappa(\alpha), r, \tau)$, where i identifies the agent, κ specifies the controller (parameterized by α) that executes in region r for τ time.

Definition 2.2 (Spatio-temporal MDL Language)**.** Let $\mathcal{P}$ be a finite alphabet of

symbols that each represent a spatio-temporal MDL mode. Then the spatio-temporal MDL language is the set of all possible concatenations of elements from $\mathcal{P}$.

To make these ideas more concrete, consider the following example string:

$$(i, \kappa_1(\alpha_1), r_1, \tau_1)(i, \kappa_2(\alpha_2), r_1, \tau_2).$$

Agent-i must complete the motion κ_1, scaled by α_1, within region r_1 until time τ_1. (Note here that even though κ_j is a function of x^i, t, and α_j, we specify it symbolically through the α_j dependency alone.) Once this mode terminates, the second mode will execute κ_2 with scaling α_2, also in region r_1, until τ_2, which in this case signals the end of the motion sequence. When this string executes on the system dynamics (9), it creates hybrid dynamics according to:

$$\dot{x}^i = \begin{cases} f(x^i, t, \kappa_1(x^i, t, \alpha_1)), \ t \in [t_0, \tau_1) \\ f(x^i, t, \kappa_2(x^i, t, \alpha_2)), \ t \in [\tau_1, \tau_2] \end{cases}.$$

Furthermore, we can visualize the operation of this hybrid system with the illustration in Figure 5.

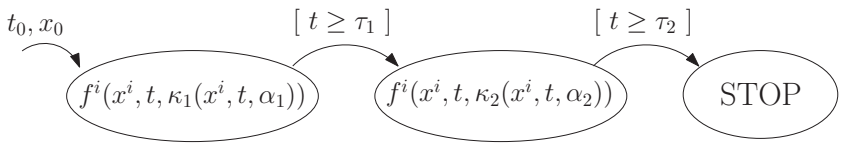


Figure 5: An illustration of the hybrid system described by the MDL string $(i, \kappa_1(\alpha_1), r_1, \tau_1)(i, \kappa_2(\alpha_2), r_1, \tau_2)$. Note the guard condition for switching between modes is the system time, t, passing the nominal switch-time τ_1. Furthermore, the system finishes execution once $t \geqslant \tau_2$.

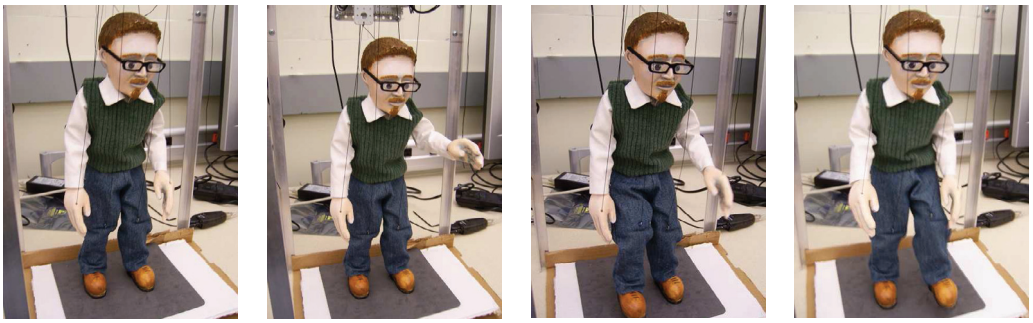
2.1.1 Languages for Puppet Plays

Puppet plays are written in a special script that enables the specification of puppet motion that must be choreographed with music and other puppets. Each line in a puppet play combines the agents involved, their motions, and the timing and spatial requirements [6, 23]. We take the important elements of puppet plays and associate

them with symbols in our spatio-temporal MDL, also known as *MDLp*. For example, this excerpt from *Rainforest Adventures*[1] describes the motion for three puppets Female (`F`), Male 1 (`1`) and Male 2 (`2`):

```
4. F 1 2 fly up and stay and drop fast
5. F hops in place, 1 hops 4 SR turns hops 4 SL
```

The left column of the script displays the *count* number, which denotes the timing for motions for the agents listed in the line. In this case, the agents `F`, `1`, and `2` will perform several actions during the fourth count of this scene. Note that the `drop` motion is parameterized by a relative speed: `fast`. We interpret this modifier as the energy parameter, α, as defined in the prior section. Another important element of the play specification is the designated regions seen in count `5`: `SR` ("stage-right") and `SL` ("stage-left"). We use these stage descriptions as the regions in the set $\mathcal{R}$.



(a) Puppet in initial configuration. (b) Puppet in wave motion. (c) Puppet starting a walk. (d) The final step in the walk mode.

Figure 6: An image sequence of the puppet executing a *wave* followed by a *walk* mode.

Accordingly, for our puppet platform (shown in Figure 6), we can create several motions for the set $\mathcal{K}$ and break up the stage into the same regions used in puppetry. For example, an MDLp mode for "walking" could be written as $(p^1, \texttt{walk}(\alpha_1), r_2, 3)$, which is interpreted as "puppet 1 walks at speed α_1 within region 2 for 3 counts." We

[1]Courtesy of Jon Ludwig, Artistic Director of the Center for Puppetry Arts, Atlanta, Georgia, `http://www.puppet.org`.

now turn to developing a "compiler" that can accept spatio-temporal MDL strings and output modified timing (τ) and energy (α) parameters, which in turn generate optimized, executable code for our autonomous puppets.

2.2 Compiling Motion Programs

In this section we discuss the derivation of our process for generating control code from high-level, spatio-temporal MDL programs. Figure 7 illustrates the general flow of this control code generation process. In particular, we use our MDLp language to enable the specification of motion programs for a collection of autonomous puppets. We derive the necessary optimality conditions for the program's switch times and scaling parameters, which are then implemented as a gradient descent algorithm in the *MDL Compiler* block. Furthermore, we enable the compilation of MDL programs when there exist hard timing constraints between puppets that must coordinate a motion.

Figure 7: An illustration of the process of turning high-level MDL programs into executable control code.

2.2.1 Motion Description Language Compiler

Now that we have modified MDL for composing multi-agent motion programs, we focus on developing a process for tweaking the timing and scaling parameters. For instance, an undesirable MDL mode would use a control law that potentially drives a system out of its intended operational region. It would be better to adjust the timing and energy scaling of the mode so that this behavior is prevented. We approach this problem using calculus of variations to derive optimality conditions that form the basis of an MDL compiler algortihm. This algorithm accepts a nominal motion program and outputs control code based on the system dynamics, under spatio-temporal

constraints.

Say we are given a single-agent (agent-i) program with N modes over the time interval $[t_0, t_f]$, and we denote all switch time parameters as the vector $\bar{\tau}^i = [\tau_1^i \ \cdots \ \tau_{N-1}^i]$ and the scaling parameters as $\bar{\alpha}^i = [\alpha_1^i \ \cdots \ \alpha_N^i]$. Additionally, we denote the *nominal* switch times and energy parameters from our spatio-temporal MDL specification as $\hat{\tau}^i = [\hat{\tau}_1^i \ \cdots \ \hat{\tau}_{N-1}^i]$ and $\hat{\alpha}^i = [\hat{\alpha}_1^i \ \cdots \ \hat{\alpha}_N^i]$, respectively. Then the cost functional for optimizing this agent's program could take the form,

$$\min_{\bar{\tau}^i, \bar{\alpha}^i} J(\bar{\tau}^i, \bar{\alpha}^i) = \int_{t_0}^{t_f} L(x^i, t)dt + \sum_{j=1}^{N} \left(C_j(\alpha_j^i, \hat{\alpha}_j^i) + \Psi_j(x^i(\tau_j^i)) \right) + \sum_{k=1}^{N-1} \Delta_k(\tau_k^i, \hat{\tau}_k^i). \quad (10)$$

The interpretation here is that the agent has a trajectory cost, $L(x^i, t)$, associated with the execution of the motion program. Since scaling controller speed or amplitude requires more energy, we penalize the energy usage of each mode with the $C_j(\alpha_j^i, \hat{\alpha}_j^i)$ functions. In this general form, the energy penalty function could use the nominal values from the initial MDL specification, $\hat{\alpha}_j^i$. We also encode the spatial constraint for each mode through the spatial cost term, $\Psi_j(x^i(\tau_j^i))$, that penalizes the distance of the agent from the location of the specified region. Finally, to prevent large deviations of a particular switch-time τ_j^i, we add the temporal cost function $\Delta_k(\tau_k^i, \hat{\tau}_k^i)$, which uses the nominal switch-times, $\hat{\tau}_k^i$.

Assume that we construct an MDLp motion program of length N, which induces the system dynamics:

$$\dot{x}^i = \begin{cases} f(x^i, t, \kappa_1(x^i, t, \alpha_1)), \ t \in [\tau_0, \tau_1) \\ f(x^i, t, \kappa_2(x^i, t, \alpha_2)), \ t \in [\tau_1, \tau_2) \\ \vdots \\ f(x^i, t, \kappa_N(x^i, t, \alpha_N)), \ t \in [\tau_{N-1}, \tau_N] \end{cases} \quad (11)$$

where $\tau_0 := t_0$ and $\tau_N := t_f$. The following result gives us the optimality conditions necessary to "compile" this motion program.

Theorem 2.1 (First Order Necessary Optimality Conditions)**.** *Given a function* $x(t) \in \mathbb{R}^n$, $\alpha \in \mathbb{R}^+$, *and* $t \in \mathbb{R}^+$. *The first order necessary conditions for minimizing the cost functional* (10) *such that the system dynamics* (11) *are satisfied are given by*

$$\begin{aligned}
\frac{\partial J}{\partial \tau_k^i} &= \lambda^i(\tau_k^{i-})^T f_k(x(\tau_k^i), \alpha_k^i) - \lambda^i(\tau_k^{i+})^T f_{k+1}(x^i(\tau_k^i), \alpha_{k+1}^i) + \frac{\partial \Delta_k^i}{\partial \tau_k^i} \\
&+ \frac{\partial \Psi_k}{\partial x^i}^T f_{k+1}^i(x^i(\tau_k^i), \alpha_{k+1}^i) = 0, k = 1, \cdots, N-1 \qquad (12) \\
\frac{\partial J}{\partial \alpha_k^i} &= \mu^i(\tau_{k-1}^{i+}) = 0, \ k = 1, \cdots, N
\end{aligned}$$

where $f_k(x^i(t), \alpha_k^i)$ *denotes* $f(x^i, t, \kappa_k(x^i, t, \alpha_k^i))$ *and* τ_k^{i-} *and* τ_k^{i+} *are the left and right limits, respectively. Additionally, the discontinuous co-states* $\lambda^i(t) \in \mathbb{R}^n$, $\mu^i(t) \in \mathbb{R}$ *satisfy the co-state dynamics,*

$$\begin{aligned}
\dot{\lambda}^i(t)^T &= -\frac{\partial J}{\partial x^i}^T - \lambda^i(t)^T \frac{\partial f_k}{\partial x^i} \\
\dot{\mu}^i(t) &= \lambda^i(t)^T \frac{\partial f_k}{\partial \alpha_k^i}
\end{aligned}$$

with $t \in [\tau_{k-1}, \tau_k]$ *and boundary conditions,*

$$\begin{aligned}
\lambda^i(\tau_N^i) &= \frac{\partial \Psi_N}{\partial x^i}(x^i(\tau_N^i)) \\
\mu^i(\tau_N^i) &= \frac{\partial C_N}{\partial \alpha_N^i} \\
\lambda^i(\tau_k^{i-}) &= \lambda^i(\tau_k^{i+}) + \frac{\partial \Psi_k}{\partial x^i}(x^i(\tau_k^i)) \\
\mu^i(\tau_k^{i-}) &= \frac{\partial C_k}{\partial \alpha_k^i}
\end{aligned}$$

for $k = 1, \cdots, N-1$ *and where we let we let* $\tau_N = t_f$ *and* $\tau_0 = t_0$.

Proof 2.1. Let $\bar{\tau} := [\tau_1 \ \tau_2 \cdots \tau_{N-1}]$ and $\bar{\alpha} := [\alpha_1 \ \alpha_2 \cdots \alpha_N]$ and without loss of generality, we drop the agent identifier, i, in the cost functional terms and system dynamics. To get the first order necessary optimality conditions for our minimization problem, we need to find the derivative of (10) with respect to the timing ($\bar{\tau}$) and energy ($\bar{\alpha}$) parameters and set it equal to 0, i.e.

$$\lim_{\epsilon \to 0} \frac{\tilde{J}_\epsilon - \tilde{J}}{\epsilon} = 0,$$

where $\tilde{J}$ is the Lagrangian and $\tilde{J}_\epsilon$ is the perturbed Lagrangian. Given the cost functional (10) and system dynamics (11), we form the Lagrangian:

$$\begin{aligned}\tilde{J}(\bar{\tau},\bar{\alpha}) &= \sum_{k=1}^{N}\int_{\tau_{k-1}}^{\tau_k} L(x,t)+\lambda_k^T(f_k(x,\alpha_k)-\dot{x})\ dt \\ &+ \sum_{j=1}^{N}\big(C_j(\alpha_j^i)+\Psi_j(x^i(\tau_j^i))\big)+\sum_{k=1}^{N-1}\Delta_k(\tau_k^i).\end{aligned}$$

Then, perturbing the timing and energy parameters, $\tau_k \rightarrow \tau_k+\epsilon\theta_k$ and $\alpha_k \rightarrow \alpha_k+\epsilon a_k$, results in the varied Lagrangian term:

$$\begin{aligned}&\tilde{J}_\epsilon(\bar{\tau}+\epsilon\bar{\theta},\bar{\alpha}+\epsilon\bar{a}) = \int_{\tau_0}^{\tau_1} L(x+\epsilon\eta,t)+\lambda_1^T(f_1(x+\epsilon\eta,\alpha_1+\epsilon a_1))-\dot{x}-\epsilon\dot{\eta})\ dt \\ &+ \int_{\tau_1}^{\tau_1+\epsilon\theta_1} L(x+\epsilon\eta,t)+\lambda_1^T(f_1(x+\epsilon\eta,\alpha_1+\epsilon a_1))-\dot{x}-\epsilon\dot{\eta})\ dt \\ &+ \int_{\tau_1+\epsilon\theta_1}^{\tau_2} L(x+\epsilon\eta,t)+\lambda_2^T(f_2(x+\epsilon\eta,\alpha_2+\epsilon a_2))-\dot{x}-\epsilon\dot{\eta})\ dt \\ &+ \cdots \\ &+ \int_{\tau_{N-1}+\epsilon\theta_{N-1}}^{\tau_N} L(x+\epsilon\eta,t)+\lambda_{N-1}^T(f_{N-1}(x+\epsilon\eta,\alpha_{N-1}+\epsilon a_{N-1}))-\dot{x}-\epsilon\dot{\eta})\ dt \\ &+ \sum_{j=1}^{N} C_j(\alpha_j+\epsilon a_j)+\sum_{k=1}^{N-1}\big(\Psi_k(x(\tau_k+\epsilon\theta_k)+\epsilon\eta(\tau_k+\epsilon\theta_k))+\Delta_k(\tau_k+\epsilon\theta_k)\big) \\ &+ \Psi_N(x(\tau_N)+\epsilon\eta(\tau_N)).\end{aligned}$$

Applying Taylor's expansion to all perturbed terms, evaluating the perturbed integrals using the mean-value theorem, and finally applying $\lim_{\epsilon\rightarrow 0}\frac{\tilde{J}_\epsilon-\tilde{J}}{\epsilon}$ results in the following equation:

$$
\begin{aligned}
\lim_{\epsilon\to 0}\frac{\tilde{J}_\epsilon-\tilde{J}}{\epsilon} &= \int_{t_0}^{\tau_1}\frac{\partial L}{\partial x}^T\eta+\lambda_1^T(\frac{\partial f_1}{\partial x}\eta+\frac{\partial f_1}{\partial \alpha_1}a_1-\dot{\eta})\ dt \\
&+ \int_{\tau_1}^{\tau_2}\frac{\partial L}{\partial x}^T\eta+\lambda_2^T(\frac{\partial f_2}{\partial x}\eta+\frac{\partial f_2}{\partial \alpha_2}a_2-\dot{\eta})\ dt \\
&+ \cdots \\
&+ \int_{\tau_{N-1}}^{t_f}\frac{\partial L}{\partial x}^T\eta+\lambda_N^T(\frac{\partial f_N}{\partial x}\eta+\frac{\partial f_N}{\partial \alpha_N}a_N-\dot{\eta})\ dt \\
&+ \sum_{k=1}^{N-1}\big(\lambda(\tau_k^-)^T f_k(x(\tau_k),\alpha_k)-\lambda(\tau_k^+)^T f_{k+1}(x(\tau_k),\alpha_{k+1})+\frac{\partial \Delta_k}{\partial \tau_k}\big)\theta_k \\
&+ \sum_{j=1}^{N}\frac{\partial C_j}{\partial \alpha_j}a_j+\sum_{k=1}^{N-1}\frac{\partial \Psi_k}{\partial x}^T(\eta(\tau_k)+\frac{\partial x}{\partial \tau_k}\theta_k)+\frac{\partial \Psi_N}{\partial x}^T\eta(t_f)=0.
\end{aligned}
\tag{13}
$$

Then, integration by parts is used to reformulate the integrals involving $\dot{\eta}$:

$$
-\int_{\tau_{k-1}}^{\tau_k}\lambda_k\dot{\eta}\ dt=-\big(\lambda_k\eta\Big|_{\tau_{k-1}}^{\tau_k}-\int_{\tau_{k-1}}^{\tau_k}\dot{\lambda}_k\eta\ dt\big)
$$

where $k=1,\cdots,N$. Inserting these terms into the integral terms of (13) results in

$$
\begin{aligned}
&\lim_{\epsilon\to 0}\frac{\tilde{J}_\epsilon-\tilde{J}}{\epsilon}= \\
&+ \int_{\tau_0}^{\tau_1}\big(\frac{\partial L}{\partial x}^T+\lambda_1^T\frac{\partial f_1}{\partial x}+\lambda_1^T\frac{\partial f_1}{\partial \alpha_1}a_1+\dot{\lambda}_1^T\big)\eta\ dt-\lambda_1^T(\tau_1)\eta(\tau_1)+\lambda_1(t_0)^T\eta(t_0) \\
&+ \int_{\tau_1}^{\tau_2}\big(\frac{\partial L}{\partial x}^T+\lambda_2^T\frac{\partial f_2}{\partial x}+\lambda_2^T\frac{\partial f_2}{\partial \alpha_2}a_2+\dot{\lambda}_2^T\big)\eta\ dt-\lambda_2^T(\tau_2)\eta(\tau_2)+\lambda_2^T(\tau_1)\eta(\tau_1) \\
&+ \cdots \\
&+ \int_{\tau_{N-1}}^{\tau_n}\big(\frac{\partial L}{\partial x}^T+\lambda_N^T\frac{\partial f_N}{\partial x}+\lambda_N^T\frac{\partial f_N}{\partial \alpha_N}a_N+\dot{\lambda}_N^T\big)\eta\ dt-\lambda_N(\tau_N)^T\eta(\tau_N)+\lambda_2(\tau_{N-1})^T\eta(\tau_{N-1}) \\
&+ \sum_{k=1}^{N-1}\big(\lambda_k(\tau_k)^T f_k(x(\tau_k),\alpha_k)-\lambda_{k+1}(\tau_k)^T f_{k+1}(x(\tau_k),\alpha_{k+1})+\frac{\partial \Psi_k}{\partial x}^T\frac{\partial x}{\partial \tau_k}+\frac{\partial \Delta_k}{\partial \tau_k}\big)\theta_k \\
&+ \sum_{j=1}^{N}\frac{\partial C_j}{\partial \alpha_j}a_j+\sum_{k=1}^{N-1}\frac{\partial \Psi_k}{\partial x}^T\eta(\tau_k)+\frac{\partial \Psi_N}{\partial x}^T\eta(\tau_N)=0.
\end{aligned}
\tag{14}
$$

Now that the $\dot{\eta}$ terms have been removed in (14), the common terms related to the different variations are collected, i.e. terms multiplied by θ_k or a_k. First, optimality

conditions for $\bar{\tau}$ take the form:

$$\frac{\partial J}{\partial \tau_k} = \lambda_k(\tau_k)^T f_k(x(\tau_k), \alpha_k) - \lambda_{k+1}(\tau_k)^T f_{k+1}(x(\tau_k), \alpha_{k+1}) + \frac{\partial \Delta_k}{\partial \tau_k} + \frac{\partial \Psi_k}{\partial x} f_k(x(\tau_k), \alpha_k) = 0 \tag{15}$$

with $k = 1, \cdots, N-1$, and the substitution $\frac{\partial x}{\partial \tau_k} = f_k(x(\tau_k), \alpha_k)$ is made in equation (14). The optimality conditions for $\bar{\alpha}$ have the form:

$$\frac{\partial J}{\partial \alpha_j} = \frac{\partial C_j}{\partial \alpha_j} + \int_{\tau_j}^{\tau_{j+1}} \lambda_j^T \frac{\partial f_j}{\partial \alpha_j} \, dt = 0 \tag{16}$$

with $j = 1, \cdots, N$. Taking these terms out of (14) results in:

$$\begin{aligned} \lim_{\epsilon \to 0} \frac{\tilde{J}_\epsilon - \tilde{J}}{\epsilon} &= \sum_{k=1}^{N} \int_{\tau_{k-1}}^{\tau_k} \big(\frac{\partial L}{\partial x}^T + \lambda_k^T \frac{\partial f_k}{\partial x} + \dot{\lambda}_k^T \big) \eta \, dt \\ &+ \sum_{k=1}^{N-1} \big(-\lambda_k(\tau_k)^T + \lambda_{k+1}(\tau_k)^T + \frac{\partial \Psi_k}{\partial x}^T \big) \eta(\tau_k) + \big(-\lambda_N(t_f)^T + \frac{\partial \Psi_N}{\partial x}^T \big) \eta(t_f) \\ &= 0. \end{aligned} \tag{17}$$

Note that the goal from this point in the derivation is to make all η trajectories "disappear," so that we get a derivative for the functional (10). Since $\eta(t)$ is an arbitrary trajectory we may set any terms multiplying $\eta(t)$ eequal to 0 In other words:

$$\begin{aligned} & \sum_{k=1}^{N} \int_{\tau_{k-1}}^{\tau_k} \big(\frac{\partial L}{\partial x}^T + \lambda_k^T \frac{\partial f_k}{\partial x} + \dot{\lambda}_k^T \big) \eta \, dt = 0 \\ \Rightarrow \quad & \dot{\lambda}_k^T = -\frac{\partial L}{\partial x}^T - \lambda_k^T \frac{\partial f_k}{\partial x}, \ t \in [\tau_{k-1}, \tau_k] \end{aligned} \tag{18}$$

for $k = 1, \cdots, N$, yields a dynamical equation for the co-state, λ_k. Additionally, the boundary conditions for these dynamical equations come from:

$$\begin{aligned} & \sum_{k=1}^{N-1} \big(-\lambda_k(\tau_k)^T + \lambda_{k+1}(\tau_k)^T + \frac{\partial \Psi_k}{\partial x}^T \big) \eta(\tau_k) + \big(-\lambda_N(t_f)^T + \frac{\partial \Psi_N}{\partial x}^T \big) \eta(\tau_N) \\ \Rightarrow \quad & \lambda_N(t_f) = \frac{\partial \Psi_N}{\partial x} \\ & \lambda_k(\tau_k) = \lambda_{k+1}(\tau_k) + \frac{\partial \Psi_k}{\partial x} \end{aligned} \tag{19}$$

for $k = 1, \cdots, N-1$. Note that the co-state equations can be considered one single co-state that is *discontinuous* at the switch times, τ_k. Therefore, using the dynamic equations in (18) and the boundary conditions in (19), the following represents the evolution of the backwards, discontinuous co-state $\lambda(t)$:

$$\begin{aligned}
&\dot{\lambda}(t)^T = -\frac{\partial J}{\partial x}^T - \lambda(t)^T \frac{\partial f_k}{\partial x},\ t \in [\tau_{k-1}, \tau_k] \\
&\lambda(\tau_N) = \frac{\partial \Psi_N}{\partial x}(x(\tau_N)) \\
&\lambda(\tau_k^-) = \lambda(\tau_k^+) + \frac{\partial \Psi_k}{\partial x}(x(\tau_k)) \\
&k = 1, \cdots, N-1.
\end{aligned} \tag{20}$$

where $-$ and $+$ denote left and right limits, respectively. Furthermore, it is computationally efficient to reformulate the $\frac{\partial J}{\partial \bar{\alpha}}$ conditions in (16) as a backwards differential equation. Considering (16), define the co-state μ as:

$$\mu(\tau_j^+) := \frac{\partial J}{\partial \alpha_j} = \frac{\partial C_j}{\partial \alpha_j} + \int_{\tau_{j-1}}^{\tau_j} \lambda_j^T \frac{\partial f_j}{\partial \alpha_j}\, dt,\ j = 1, \cdots, N. \tag{21}$$

Then, using the standard solution of ordinary differential equations, the dynamical equation of μ is:

$$\begin{aligned}
&\dot{\mu}(t) = \lambda(t)^T \frac{\partial f_j}{\partial \alpha_j},\ t \in [\tau_{j-1}, \tau_j] \\
&\mu(\tau_N) = \frac{\partial C}{\partial \alpha_N} \\
&\mu(\tau_j^-) = \frac{\partial C}{\partial \alpha_j} \\
&j = 1, \cdots, N.
\end{aligned} \tag{22}$$

Finally, combing the optimality conditions (15) and (16), as well as the backwards co-state equations (20) and (22) completes the proof. □

Theorem 2.1 yields the core component of the MDL compiler, which is implemented numerically using a gradient descent search. Algorithm 2.1 produces a *locally* optimal solution to (10) that yields a modified MDLp program with optimized timing and energy parameters. The algorithm accepts an MDLp string of length N and

initializes the timing and energy parameters as the arrays: $\bar{\tau}(0)$ and $\bar{\alpha}(0)$. Then, it enters a loop that computes the forward integration of the state, $x(t)$, and backward integration of the co-states, $\lambda(t), \mu(t)$. Additionally, it executes gradient descent on the timing and energy variables at each step. This gradient descent uses fixed step-sizes: $\gamma_\tau, \gamma_\alpha \in (0, 1)$. These computations continue until the gradients of the cost,

$$\nabla J := \begin{bmatrix} \frac{\partial J}{\partial \bar{\tau}} \\ \frac{\partial J}{\partial \bar{\alpha}} \end{bmatrix}$$

are within some bound $\delta > 0$. Once this condition is reached (or the maximum number of iterations, k_{max}), the k^{th} value of $\bar{\tau}$ and $\bar{\alpha}$ are returned as the optimized parameters.

Algorithm 2.1 MDL Gradient Descent

Given $(p^i, \kappa_1(\alpha_1), r_1, \tau_1)(p^i, \kappa_2(\alpha_2), r_2, \tau_2) \cdots (p^i, \kappa_N(\alpha_N), r_N, \tau_N)$
Initialize $\bar{\tau}^i(0) = [\tau_1 \ \tau_2 \ \cdots \ \tau_{N-1}]^T$, $\bar{\alpha}^i(0) = [\alpha_1 \ \alpha_2 \ \cdots \ \alpha_N]^T$
$k = 0$
while $\|\nabla J\| \leq \delta$ `AND` $k < k_{max}$ **do**
  Compute $x^i(t)$ forward with $\bar{\tau}^i(k)$, $\bar{\alpha}^i(k)$
  Compute $\lambda^i(t)$, $\mu^i(t)$ backwards
  $\bar{\tau}^i$ Descent: $\bar{\tau}^i(k+1) = \bar{\tau}^i(k) - \gamma_\tau \frac{\partial J}{\partial \bar{\tau}}^T(\bar{\tau}^i(k))$
  $\bar{\alpha}^i$ Descent: $\bar{\alpha}^i(k+1) = \bar{\alpha}^i(k) - \gamma_\alpha \frac{\partial J}{\partial \bar{\alpha}}^T(\bar{\alpha}^i(k))$
  if $\|\nabla J\| \leq \delta$ **then**
    $\bar{\tau}^{i,\star} = \bar{\tau}^i(k)$
    $\bar{\alpha}^{i,\star} = \bar{\alpha}^i(k)$
  end if
  $k = k + 1$
end while
return $\bar{\tau}^{i,\star}$, $\bar{\alpha}^{i,\star}$

2.2.2 Constrained Timing Coordination

The result of Theorem 2.1 yields conditions for optimizing MDL strings of multiple agents, considering each agent's dynamics, timing, energy, and spatial costs. However, many systems require *hard* timing constraints, such as terminating one particular mode *before* some other agent's mode completes. In a puppet play, missing these

timing constraints may lead to benign issues, such as awkward character placement, to serious problems, such as collisions and string tangling. This section considers generating optimized MDL programs under timing inequality constraints by distributing the constraint among the agents.

In this problem, we assume that the motion program has m agents, each operating under their own dynamics. Additionally, each agent switches between C_i control modes, with the terminal time denoted by $t_f := \tau_{C_i},\ i \in \mathcal{M}$, where $\mathcal{M}$ is the same index set from Section 2.1. These agents, individually, have dynamics of the form (11), moreover, we define the global cost functional as

$$J(\bar{\tau}^1, \ldots, \bar{\tau}^m) = \sum_{i=1}^{m} J^i(\bar{\tau}^i) \tag{23}$$

where $J^i(\bar{\tau}^i)$ is the cost functional from equation 10. (Note that we exclude $\bar{\alpha}$ in this notation since we are not considering the coordination of $\bar{\alpha}$ among agents.) To illustrate the way in which the temporal constraints show up, we assume, without loss of generality, that the temporal constraint only affects the d^{th} switch for systems j and k, where $j, k \in \mathcal{M}$, as $\tau_d^j - \tau_d^k \leqslant 0$. This minimization formulation results in a *separable* optimization problem, since the function to be minimized (23) and the timing constraint depend additively on their domains [49].

This optimization problem can be solved by augmenting the cost with a Lagrangian term $\nu(\tau_d^j - \tau_d^k)$, and then jointly solving it across all the switching times for all the puppets. However, we do not want to use this centralized solution, since the ultimate goal is to have several autonomous agents (or in this case, puppets) optimize their plays in a decentralized fashion. Since we have already noted that the problem is separable we can break up the solution process. We specifically choose the approach known as *team theory*, recently explored in [52]. (Note that the details given highlight the *application* of team theory to the problem of distributed timing control as it pertains to the robotic marionette application.)

Say that puppets j and k ($j \neq k$) are temporally constrained via the d^{th} switch as $\tau_d^j - \tau_d^k \leq 0$. The constrained problem becomes

$$L(\tau_d^j, \tau_d^k, \nu) = J^j(\tau_d^j) + J^k(\tau_d^k) + \nu(\tau_d^j - \tau_d^k) \tag{24}$$

where we have assumed, without loss of generality, that the only control parameters are τ_d^j and τ_d^k, and the other switching times are considered to be fixed. It should be noted that the cost functionals are decoupled, i.e. cost J^j depends *only* on system j's dynamics. Therefore, taking the derivative of the Lagrangian with respect to ν results in the expression,

$$\frac{\partial L}{\partial \nu} = \tau_d^j - \tau_d^k,$$

in combination with the previously defined gradient expressions for the switching times.

Algorithmically, this formulation is interesting since the dual problem becomes $g^\star = \max_\nu g(\nu),\ \nu \geqslant 0$, where

$$g(\nu) = \min_{\tau_d^j, \tau_d^k} \{J^j(\tau_d^j) + J^k(\tau_d^k) + \nu(\tau_d^j - \tau_d^k)\}. \tag{25}$$

We would like to find a *local* minimum, since a global minimum may actually lead to behavior that is undesired by the original motion program specification, i.e. the removal of a mode in the sequence.

An inefficient solution to this max-min problem would apply gradient descent on the minimization problem (25) and follow with a *single* gradient ascent step for $\max_\nu g(\nu)$, and repeat until a solution is found. However, since we already know that this max-min problem is separable, we can solve it using a saddle-point search algorithm, known as *Uzawa's algorithm* [4]. The Uzawa algorithm allows the descent *and* the ascent steps to be performed simultaneously. Consequently, we use a gradient descent for the switch times, and a gradient ascent for the Lagrange multiplier ν allowing us to decouple the numerical solution process and let the networking aspect

be reflected only through the update of the multiplier, as was done in [52]. In fact, if we let

$$\dot{\tau}_d^j = -\frac{\partial J^j}{\partial \tau_d^j} - \nu$$
$$\dot{\tau}_d^k = -\frac{\partial J^k}{\partial \tau_d^k} + \nu$$
$$\dot{\nu} = \tau_d^j - \tau_d^k$$

all that needs to be propagated between the two systems is the current value of the Lagrange multiplier ν. This observation in [52] leads us to a general architecture for solving networked, switched-time optimization problems involving pairwise timing constraints, as shown in Figure 8.

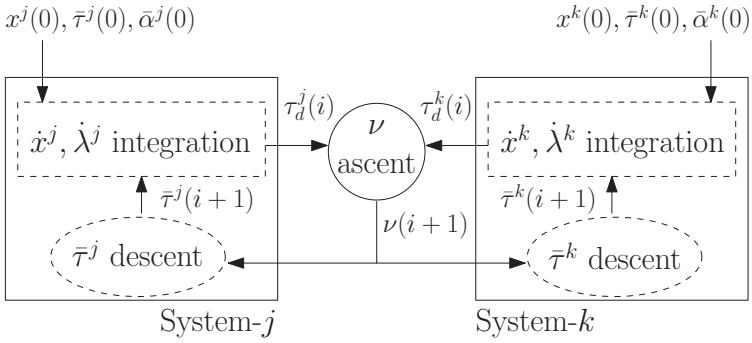


Figure 8: This figure shows how information propagates between the two subsystems (puppets) in order to solve the networked timing problem. The initial values for system j are denoted $\bar{\tau}^j(0)$, $\bar{\alpha}^j(0)$ and similarly for system k.

This numerical architecture lets us optimize the switch times individually at each algorithm iteration, denoted by the index i. First, we initialize both systems with feasible switch times and scaling parameters based on a play of length N. These values we denote with the arrays, $\bar{\tau}^j(0) = [\tau_1^j(0) \ \cdots \ \tau_{N-1}^j(0)]$ and $\bar{\alpha}^j(0) = [\alpha_1^j(0) \ \cdots \ \alpha_N^j(0)]$, respectively. Then each agent performs the forward-backward integration of their systems (x^j, x^k) and their associated co-states (λ^j, λ^k). In parallel to those integrations, the ν state is incremented using the current values for the switch times. These values are then passed to the individual systems so that they can take their gradient descent steps with the new ν values.

2.3 Simulation and Experimental Results

The theoretical development discussed in Section 2.2 provides a systematic way to compile motion programs for a collection of agents with timing, energy, and spatial constraints. This section demonstrates the sucessful applications of the MDLp compiler to a single puppet system (Section 2.3.1), a group of independent puppets (Section 2.3.2), and, finally, two puppets that need to ensure their motions execute at a particular time (Section 2.3.3).

2.3.1 Example: Single Puppet Play Optimization

In order to test the MDLp compilation process, as prescribed in Algorithm 2.1, a Matlab simulation was developed with the implemented algorithm, a small library of puppet motions, and a kinematic model of the puppet. (Note that we choose a simplified model for this puppet for less intensive algorithm computations. For a deeper examination of puppet models see [31, 62].) This kinematic model is based on the geometry seen in Figure 9. The arms are rotated about the $\bar{K}$-axis ($+\bar{K}$ for left arm, $-\bar{K}$ for right arm) by motors and the joint angles are denoted as θ_l and θ_r, respectively. The arms and legs are lifted by motor rotations about the $\bar{J}$-axis, where the arm joint angles are denoted by ϕ_l and ϕ_r and the leg joint angles are denoted by ψ_l and ψ_r. Since this model of the puppet is kinematic, our dynamic equation is given by $\dot{q} = Gu$ where $q = [\theta_l\ \theta_r\ \phi_l\ \phi_r\ \psi_l\ \psi_r]^T$, G is a diagonal matrix, and u is the mode input.

For this simulation we created two open-loop controllers for the puppet: $\mathcal{K} = \{\kappa_1 = \texttt{waveLeftArm}, \kappa_2 = \texttt{walk}\}$. For example, the `walk` mode applies alternating sinusoids of the form

$$u(t) = \frac{4}{\pi}\alpha\omega_{max}(\sin(2\pi ft) + (1/3)\sin(6\pi ft)),$$

where ω_{max} is a constant maximum rotational speed of the arm and leg lifting actuators and α is the scaling parameter for the control.

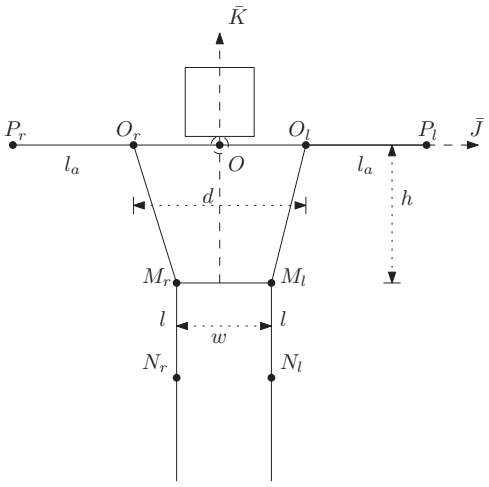


Figure 9: The geometrical configuration of the puppet under consideration. The arms have two degrees of freedom each, rotation and lift. The legs have only one degree of freedom for lifting.

Using these two controllers we construct the following play for puppet-1

$$(p^1, \kappa_1(1), r_1, 4)(p^1, \kappa_2(1), r_1, 6),$$

where the energy scaling for both modes is unity, i.e. $\alpha_1 = \alpha_2 = 1$. This simple motion program specifies that the puppet must wave its left arm for 4 seconds in region 1 and then walk in region 1 for 6 seconds. To facilitate the MDLp compilation, the following cost functional is formulated:

$$J(\tau, \alpha_1, \alpha_2) = \int_0^{t_f} q^T P q \; dt + \rho(T - \tau)^2 + w_1\alpha_1^2 + w_2\alpha_2^2, \tag{26}$$

where $t_f = 4 + 6 = 10$, P is a suitable weight matrix, and ρ, w_1, w_2 are cost weights that prescribe relative weights to deviations from the nominal switch time and motion intensity parameters. Before compiling the play using (26) as the cost, we simulate the nominal program, which generates the plot seen in Figure 10. Note that this initial trajectory results in the left arm's odd looking behavior, where the wave motion stops the left arm in "mid-air" as the walk motion begins, as seen in the simulation output of Figure 10. A more desirable trajectory would lower the left arm completely before initiating a walk.

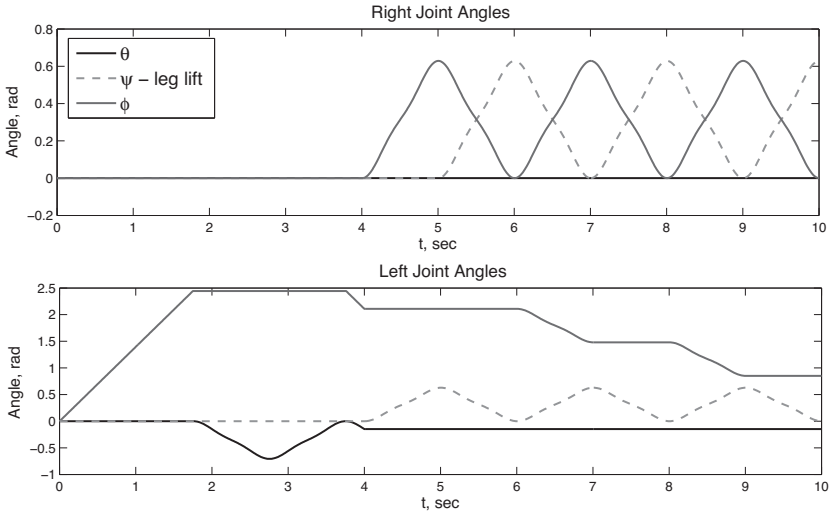


Figure 10: Original puppet joint angle trajectories.

To remedy this behavior, we defined the following weights: $\rho = 1$, $w_1 = w_2 = 5$, and

$$P = \begin{bmatrix} \mathbb{I}_4 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$

is a 6×6 matrix that penalizes only the arm angles of q and $\mathbb{I}_4$ is a 4×4 identity matrix. In other words, the arm joint angles are penalized for deviations from the puppet's "home" position, i.e. the initial arm joint angles $\theta_l = \theta_r = \phi_l = \phi_l = 0$. After an iterative, descent-based optimization algorithm has terminated, the new values become $\tau = 4.3423$, $\alpha_1 = 1.3657$, $\alpha_2 = 0.9566$, with the corresponding joint angles shown in Figure 11. This plot shows that at termination the joint angles were close to 0 before starting the walk mode, resulting in a more natural looking motion. Thus, the left arm is brought back to the side of the puppet at the new switch time, resulting in a more natural transition to the `walk` mode. Additionally, examining the total cost plot in Figure 12 reveals that the cost is indeed reduced as the gradient descent algorithm ran past 20 iterations.

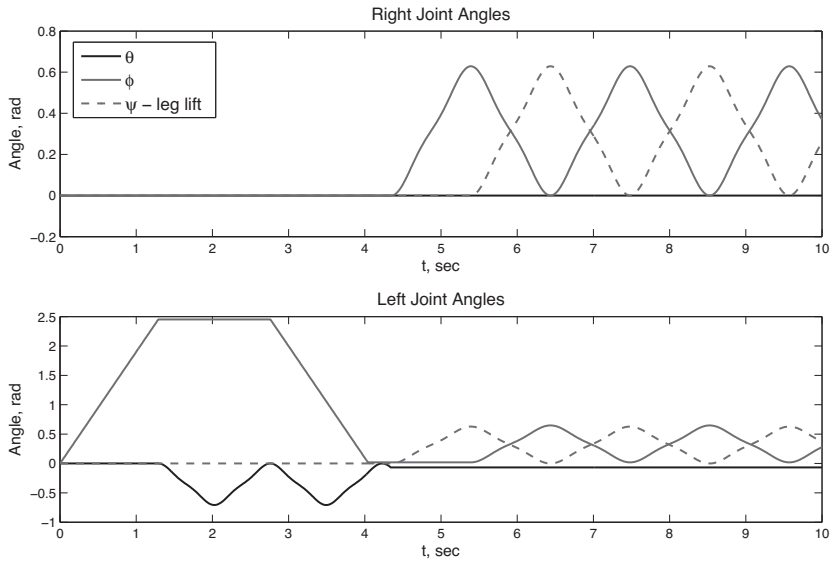


Figure 11: Optimized puppet joint angle trajectories.

2.3.2 Example: Spatial Optimization with Multiple Puppets

In this section we demonstrate the compilation of an MDLp program for multiple puppets. Although the actual puppet dynamics are quite complex, the spatial location of the puppet is modeled without taking the joint angles into account. Instead, the implemented puppet system would use planar unicycle dynamics to drive the puppet around the stage. Therefore, we model each agent's dynamics as,

$$\dot{z} = \begin{bmatrix} \alpha v \sin\gamma \\ \alpha v \cos\gamma \\ u_\gamma \end{bmatrix}.$$

The α in these dynamics is the scaling parameter discussed before and v is a constant maximum speed. Additionally, γ represents the heading angle of the puppet and is driven directly by some signal u_γ. Also, the joint angles of the arms and legs are represented, as before, by the vector $q = [\theta_r\ \phi_r\ \theta_l\ \phi_l\ \psi_r\ \psi_l]^T$, where the r and l subscripts denote right and left, respectively. We concatenate the z and q states, and denote the puppet's state as $\bar{x} = [z\ q]^T$. In this example, we want each puppet to

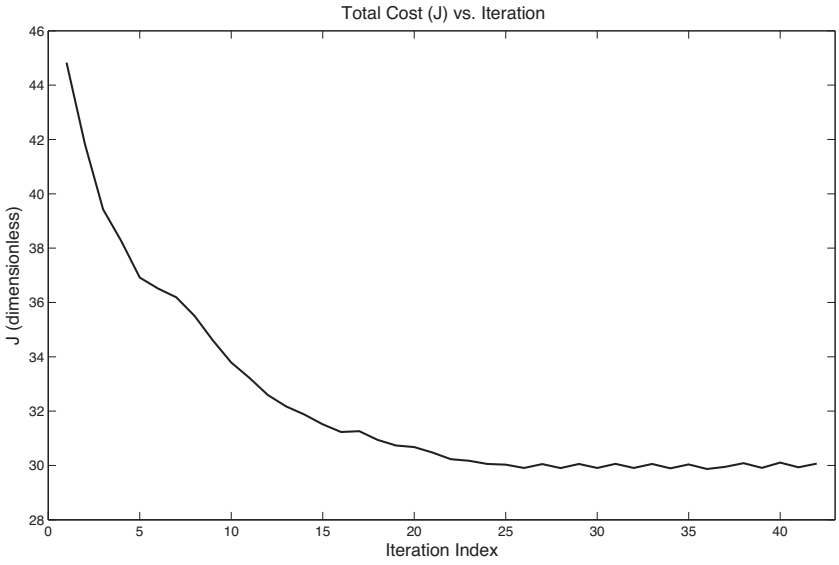


Figure 12: Plot of the total cost over simulation iterations.

stay as close to the center of their designated regions as possible; therefore, we use a quadratic cost for $L(\bar{x}, t)$ and $\Psi(\bar{x}(\tau))$ in (10). Additionally, the desired trajectory terms in these costs, denoted by $\bar{x}^d$, will depend on the regions specified in the MDLp script.

Using these cost design choices, we let $L(\bar{x}, t) = (\bar{x} - \bar{x}^d)^T P(\bar{x} - \bar{x}^d)$, where P is a 9×9 positive definite weight matrix. The other cost function that accounts for spatial penalties is $\Psi(\bar{x}(\tau))$. We define this function as,

$$\Psi_k(x(\tau_k)) = (\bar{x}(\tau_k) - \bar{x}^d(\tau_k))^T Z(\bar{x}(\tau_k) - \bar{x}^d(\tau_k)),\ k = 1, \cdots, N,$$

where $Z \succ 0$ is another weight matrix. This function is similar to $L(\bar{x}, t)$; however, its weight matrix penalizes *only* the planar position of the agent, and it is evaluated only at the switch times, τ_k. Finally, we penalize the scaling factors and time deviations in the same way as in Section 2.3.1: $C_j = w_j \alpha_j^2$, for $j = 1, \cdots, N$, and $\Delta_k(\tau_k) = \rho_k(T_k - \tau_k)^2$ for $k = 1, \cdots, N-1$.

In this example, we use the controllers κ_1 and κ_2 from the example in Section 2.3.1 and add in the new controller $\kappa_3 =$ `walkInCircles`, which causes the puppet to

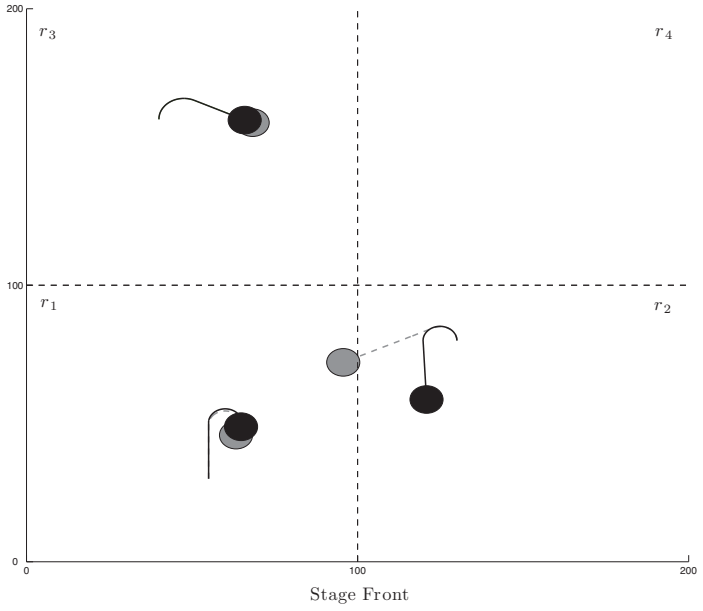


Figure 13: Image of the puppet motions before (gray) and after (black) the MDL compilation process.

walk with a circular arc. Using these controllers, we constructed the following MDLp play:

$$(p^1, \kappa_1(1.2), r_1, 2.5)(p^1, \kappa_2(1.3), r_1, 3)(p^1, \kappa_3(1), r_1, 4)$$
$$(p^2, \kappa_1(1.2), r_3, 2.5)(p^2, \kappa_3(1.5), r_3, 2)(p^2, \kappa_2(1.3), r_3, 3)$$
$$(p^3, \kappa_3(1), r_2, 2)(p^3, \kappa_2(1.5), r_2, 4).$$

The initial run of this MDLp play is illustrated by the *gray* lines and shapes in Figure 13. Note that puppets 1 and 2 (located in r_1 and r_3 in the figure) behave relatively well using their nominal plays. However, puppet 3 breaches the boundary between r_1 and r_2 while its MDLp string requires it to remain in r_2.

After running the MDL compiler on these strings, the improved runtime behavior is illustrated by the black lines and shapes in Figure 13. Puppet 3's trajectory is now within r_2, as prescribed in the original MDLp string. Also, all three puppets

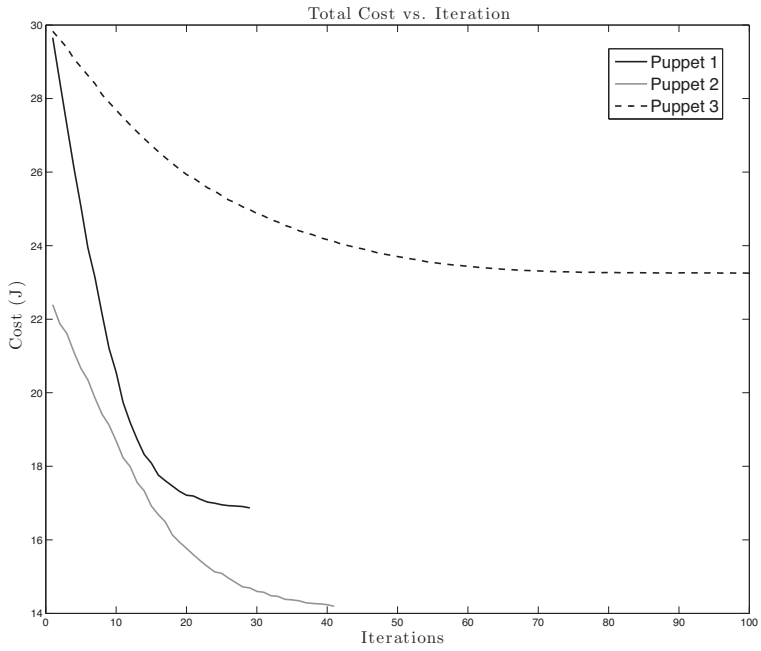


Figure 14: This figure shows the costs as a function of the MDL compiler algorithm iteration when compiling a play for three puppets with *spatial* constraints. Puppet 1 completed in 29 iterations, Puppet 2 completed in 41 iterations, and Puppet 3 took 100 iterations.

reduce their cost, as shown in Figure 14. Note that puppet 3 takes the longest, computing 100 iterations before minimizing its cost. This iteration count shows how bad the nominal program was at satisfying the cost functional (10). Additionally, our algorithm uses a conservative, fixed-step gradient descent to limit the amount of numerical error, which will slow down convergence as the derivatives (12) get closer to 0. If a dynamic step size were used (such as Armijo step-size [3]) then convergence would be faster. This work demonstrates that we can solve the problem of improving the multi-agent motion program given spatial costs. We now turn to the example of generating optimized control code under networked timing constraints.

2.3.3 Example: Time-switch Constraints for Puppetry

Using the same collection of control laws from Section 2.3.2 we define a multi-puppet play as follows,

$$(1, \kappa_1(1.2), r_1, 2.5)\ (1, \kappa_2(1.3), r_1, 3)\ (1, \kappa_3(1), r_1, 3)$$
$$(2, \kappa_1(1.2), r_3, 2.5)\ (2, \kappa_3(1.5), r_3, 3)\ (2, \kappa_2(1.3), r_3, 2.5).$$

This play uses two agents, both executing three modes with various timing requirements and scaling parameters.

In this simulation, we choose to constrain the *first* switch of each puppet, i.e. $d = 1$. If we denote $\bar{\tau}^i = [\tau_1^i\ \tau_2^i]$ as the switch times and $\bar{\alpha}^i = [\alpha_1^i\ \alpha_2^i\ \alpha_3^i]$ as the scaling parameters for puppet i, then the constrained minimization problem for these two puppets is stated as

$$\min_{\bar{\tau}^1, \bar{\tau}^2, \bar{\alpha}^1, \bar{\alpha}^2} J^1(\bar{\tau}^1, \bar{\alpha}^1) + J^2(\bar{\tau}^2, \bar{\alpha}^2)$$
$$\text{s.t. } \tau_1^2 \leqslant \tau_1^1.$$

We formulate a Lagrangian for this problem as in equation (24),

$$L(\bar{\tau}^1, \bar{\alpha}^1, \bar{\tau}^2, \bar{\alpha}^2, \nu) = J^1(\bar{\tau}^1, \bar{\alpha}^1) + J^2(\bar{\tau}^2, \bar{\alpha}^2) + \nu(\tau_1^2 - \tau_1^1),$$

and then apply the algorithm visualized in Figure 8.

Figure 15 displays the cost graphs for the two puppets after the execution of the distributed algorithm. The cost is indeed reduced for both puppets and, furthermore, Figure 16 shows that the required inequality constraint is satisfied. The optimal switch times and scaling parameters for puppet 1 are $\bar{\tau}^1 = [2.9906\ 3.0463]$ and $\bar{\alpha}^1 = [1.1903\ 1.3249\ 1.0204]$, respectively. Additionally, the results for puppet 2's parameters are $\bar{\tau}^2 = [2.9683\ 2.9157]$ and $\bar{\alpha}^2 = [1.1901\ 1.5228\ 1.3124]$.

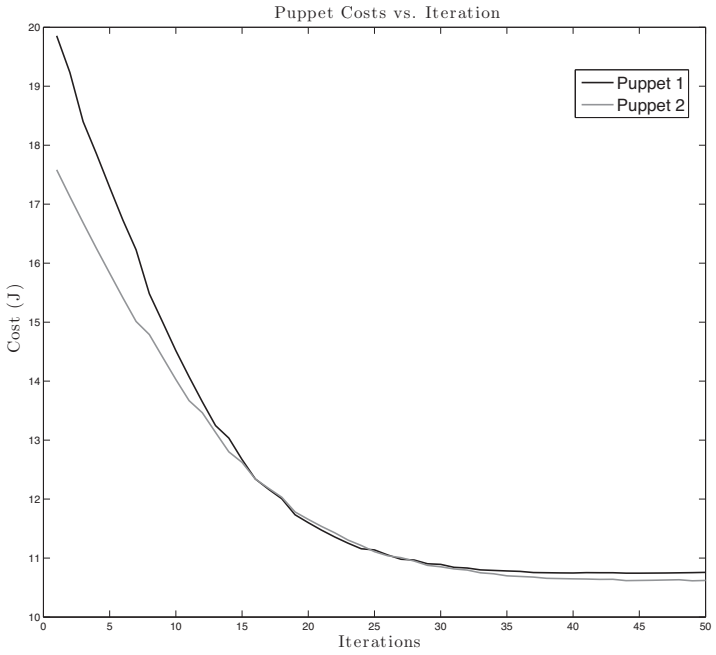


Figure 15: The cost of both puppets using the distributed switch time constraint architecture.

2.4 Conclusions

In this chapter, the problem of compiling motion programs with timing, energy, and spatial constraints was considered. A new MDL was defined that enables the construction of such motion programs. Furthermore, an optimal control problem was formulated such that each element of the MDL symbols is used in an optimization process to produce new switch times and energy scaling parameters. A gradient descent-based algorithm was designed and implemented to facilitate the automatic compilation of MDLp programs. Finally, simulation results of this algorithm demonstrate its effectiveness in automatically adjusting MDL programs for execution on target platforms.

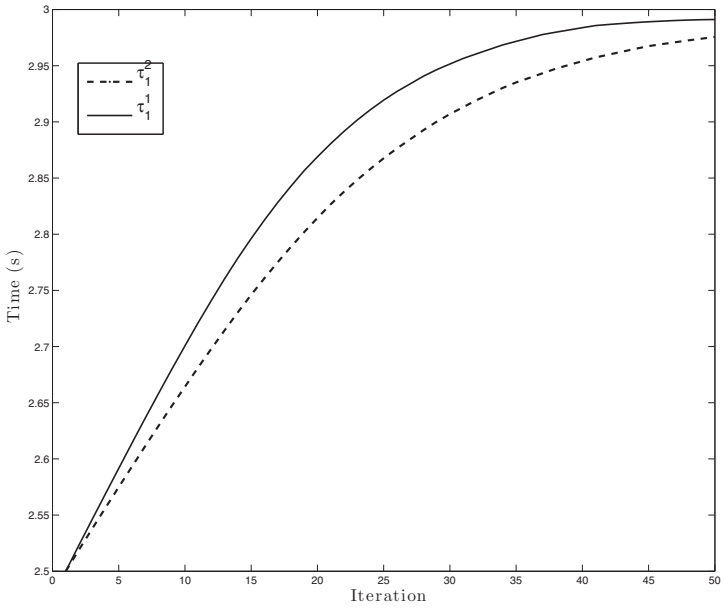


Figure 16: A graph of the constrained switch time values τ_1^2 and τ_1^1.

CHAPTER III

MULTI-AGENT MOTION PROGRAMS

In this chapter, we consider the specification of motion programs for multi-agent systems that may operate in dynamic environments, such as mobile robots or building control networks. The goal of this effort is to develop a framework that facilitates the specification and compilation of multi-agent motion programs. Furthermore, if the compilation process identifies errors in the motion program, it should automatically generate a supervisor that can prevent the execution of incorrect modes at runtime.

In Section 3.1, we describe the details of our multi-agent motion description language, called MDLn, as well as the construction of a grammar that accepts this language. Following these definitions, we develop the necessary structures and definitions to describe the MDLn compilation process and the automatic generation of MDLn supervisors in Sections 3.2 and 3.3. Finally, Section 3.4 demonstrate the MDLn specification to execution process with simulation results involving multi-robot teams.

3.1 Motion Programs for Multi-Agent Systems

What makes multi-agent mission specification conceptually different from the single agent case is that coordination and information sharing play a key role. We capture these features by modifying the dynamics for standard MDL (2) to allow for the transmission of information among a collection of N agents, with index set $\mathcal{N} = \{1, 2, \cdots, N\}$:

$$\begin{aligned} \dot{x}^i &= f^i(x^i, u^i),\ x^i \in \mathcal{X}^i \subseteq \mathbb{R}^n, u^i \in \mathcal{U}^i \\ y^i &= h^i(x^i),\ y^i \in \mathcal{Y}^i \subseteq \mathbb{R}^p \\ s^i &= g^i(x^i, y^i), s^i \in \mathcal{S}^i \subseteq \mathbb{R}^q, \end{aligned} \tag{27}$$

where $q \leqslant \dim(\mathcal{X}^i) + \dim(\mathcal{Y}^i)$. The way these entities should be understood is as follows: agent-i's dynamics are driven by its state, x^i, under the control input, u^i. The state, x^i, determines the local information produced by the agent's sensors, y^i, and, furthermore, agent-i transmits its *shareable information*, s^i, by mapping its state and sensor output onto $\mathcal{S}^i$ via the function $g^i : \mathcal{X}^i \times \mathcal{Y}^i \to \mathcal{S}^i$. (Note that this product of state and output spaces may not be needed; however, the inclusion of $\mathcal{Y}^i$ makes the environmental dependence of shared information more explicit.) This information may then be transmitted through the network to desired neighbor agents.

For example, say agent-i, which we denote as a^i, is a mobile robot with state $x^i = [x_1^i \ x_2^i \ x_3^i]^T$, where (x_1^i, x_2^i) is the Cartesian coordinate of the robot and x_3^i its orientation. Additionally, let a^i have a four sensor sonar-array, where each sonar produces two data points for each reading, i.e. $p_j^i \in \mathbb{R}^2$. Then the output vector of a^i is $y^i = [p_1^i \ p_2^i \ p_3^i \ p_4^i]^T \in \mathbb{R}^8$. If a^i plans to share its heading, x_3^i, and the forward sensor outputs, p_1^i and p_2^i, then the following shareable information vector is produced by the mapping s^i:

$$s^i = \begin{bmatrix} x_3^i \\ p_1^i \\ p_2^i \end{bmatrix}.$$

This function allows a^i to only share the information that it wishes to reveal to members of its network. However, agents do not share arbitrarily, since broadcasting the data to the network would cause unnecessary traffic.

Using the model (27) we construct a multi-agent MDL, or *MDLn*, by coupling the controller and interrupt functions from Section 1.2.2 and adding a new element for specifying the desired network information dependencies that the controller and interrupt require. This list of dependencies, called the *buddy list*, is a collection of the neighbors with which a particular agent wants to perform actions or share information. First, we assume that each agent has an "egocentric" network (denoted

$\mathcal{W}^i(t)$) of agents within its communication range at any time, t. Then, the set of agent-i's buddies are given by the following definition:

Definition 3.1 (Agent Buddies)**.** Let a^i, $i \in \mathcal{N}$, denote agent-i. The *static buddies* of a^i, denoted β_s^i, is the set of agents specified in advance of a^i's execution of its motion program. The *dynamic buddies* of a^i, denoted β_d^i, are the set of agents set by the mapping $b^i : \mathcal{W}^i \times \mathcal{Y}^i \to 2^{\mathcal{W}^i}$. Furthermore, the total set of a^i's buddies at time t is

$$\beta^i(t) = (\beta_s^i \cap \mathcal{W}^i(t)) \cup \beta_d^i(t) \tag{28}$$

Equation (28) shows that total set of agent-i's available buddies is dependent on the current available list of agents on the network at any time. From the model in equation (27) we know that each agent chooses to share their information with the vector s^i. Assume that agent-i has k buddies at time t, i.e. $\beta^i(t) = \{a^1, a^2, \cdots, a^k\}$, and each of these agents transmits their information vectors to agent-i: $s^1, s^2, \cdots, s^k$. Agent-i combines these vectors into a *locally* held vector denoted by

$$\hat{s}^i = [s^1 \ s^2 \cdots \ s^k],$$

where $\hat{s}^i \in \hat{\mathcal{S}}^i \subseteq \mathbb{R}^{kq}$. Thus, agent-$i$ can use the shared information of its "buddy agents" when making control and interrupt decisions.

Using the above definitions and variables, the control and interrupt functions from MDL are modified as follows. The control depends on the state and sensor feedback of agent-i, the information from all buddies of agent-i, and time ($\mathbb{R}^+$):

$$\kappa^i : \mathcal{X}^i \times \mathcal{Y}^i \times \hat{\mathcal{S}}^i \times \mathbb{R}^+ \to \mathcal{U}^i.$$

Additionally, the interrupt function uses the same local and shared information as

$$\xi^i : \mathcal{X}^i \times \mathcal{Y}^i \times \hat{\mathcal{S}}^i \times \mathbb{R}^+ \to \{0, 1\}.$$

Using these reformulated control and interrupt mappings, we define an MDLn mode as follows:

Definition 3.2 (MDLn Mode)**.** An *MDLn mode* is the tuple $(a^i, \kappa^i, \xi^i, \beta^i)$, where a^i identifies the agent, κ^i represents the currently active control mapping, ξ^i is the current interrupt mapping, and β^i is the current buddy list required for execution. Furthermore, we denote the k^{th} MDLn mode of agent-i as $\sigma_k^i := (a^i, \kappa_k^i, \xi_k^i, \beta_k^i)$.

The *MDLn language* is the set of all possible concatenations of these MDLn modes.

3.1.1 Agent Interaction Rules

Many multi-agent systems require that the agents be assigned different *roles*, which in turn affect the type of actions the agents can perform or information they can obtain. In addition to the convoy protection task discussed in Section 1.1, one can imagine other leader-follower or team-based applications where agents partition the network into different command hierarchies. We use roles to specify the network hierarchy of the agents involved.

Definition 3.3 (Agent Roles)**.** The role of an agent is a static value resulting from the mapping $r : \mathcal{N} \rightarrow \mathcal{R}$, where $\mathcal{R}$ is a set with total order.

Furthermore, agents use these roles to determine the members of the network with which they can exchange information via the following set of rules:

Definition 3.4 (Agent Interaction Rules)**.** Given two agents, a^i and a^j,

R1: If $r(i) > r(j)$ then a^i is able to receive shared information from a^j.

R2: If $r(i) = r(j)$ then a^i and a^j may share information with each other.

R3: If $r(i) > r(j)$ and $a^j \in \beta^i$ then a^i and a^j may share information with each other.

We interpret these rules as follows: R1 states that if the value of a^i's role is higher than the role value of a^j then a^i may pull any shareable information from a^j without restriction. R2 describes the case when a^i and a^j share the same role value, and hence can exchange their shareable information without restriction. Finally, R3 provides an

exceptional case where a^j has a lower role than a^i; however, a^i already plans to work with a^j since a^j is listed in its own buddy list, β^i.

3.1.2 MDLn Example

To make the MDLn language definition more concrete, we consider an example MDLn string involving three robot agents, a^1, a^2, and a^3 with role assignments $r(a^1) = 1$ and $r(a^2) = r(a^3) = 0$. In this configuration, a^1 has a higher role and can be considered the "leader" of a^2 and a^3. Furthermore, let these robots have three implemented controllers: κ_{gtg} sends the robot toward a goal point (where the gtg abbreviation means `GoToGoal`), κ_{avoid} causes the robot to avoid an obstacle, and κ_f = `Follow` causes a robot to follow another. Additionally, we define the robot's interrupt conditions, ξ_{obs}, which transitions to 1 when an obstacle is detected, and ξ_{clr} (where clr stands for `Clear`), which triggers when the sensors detect no obstacles One example of an MDLn string using these controllers and interrupts is:

$$(a^2, \kappa_{gtg}, \xi_{obs}, \{\})(a^1, \kappa_f, \xi_{obs}, \{a^2\})(a^3, \kappa_f, \xi_{obs}, \{a^1\}). \tag{29}$$

This string tells a^2 to head toward a fixed goal location until it detects an obstacle, and consequently terminates operation. The second mode in the program instructs a^1 to follow a^2, since its buddy list is $\beta^1 = \{a^2\}$. Additionally, the third mode directs a^3 to follow a^1 due to its list, $\beta^3 = \{a^1\}$. We can visualize the sharing of information among the agents with the diagram in Figure 17. Agent-1 is able to execute its κ_f controller since its role is valued higher than agent-2 and is granted access to the information according to interaction rule R1. Unfortunately, agent-3 will not be able to follow agent-1 since its network dependency violates R1. This simple string reveals the need for an MDLn "compiler" that can not only parse the high-level MDLn language, but also determine whether an MDLn program can be executed correctly.

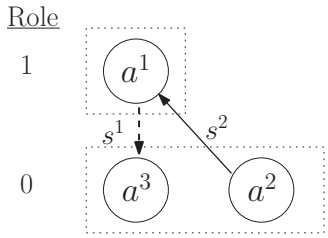


Figure 17: An illustration of the available shared information among the three agents using the MDLn string in (29). Agent-1 is able to get agent-2's data (solid line); however, agent-3 is denied the data from agent-1 (dashed line).

3.1.3 Parser

The MDLn language presented thus far provides a method for us to specify a string of motions for a set of agents. However, the strings alone do not provide enough information for determining whether the agents can execute their given programs. To do so, we need to construct a grammar that combines the MDLn mode structure in Definition 3.2 with the role specifications in Definition 3.3. A parser based on this grammar outputs MDLn strings, which each implement the controllers described by the motion program, and the associated role information of the agents.

According to the standard definition of a grammar [29], we define our MDLn grammar as:

Definition 3.5 (MDLn Grammar)**.** The *MDLn grammar* is the tuple

$$G = (L, T, \Pi, \omega)$$

where $L = \{O, R, I, M\}$ is the set of *non-terminals*, $T = \{r, k, z, b, (,)\}$ is the set of terminals, Π is the set of production rules:

$$\begin{aligned} O &\rightarrow R^{\star}\ M^{+} \\ R &\rightarrow I\ =\ r \\ M &\rightarrow (I\ k\ z\ \{I^{\star}\}). \end{aligned} \tag{30}$$

Additionally, the start symbol for the MDLn grammar is $\omega = O$.

The first line of (30) shows the start symbol, $\omega = O$, which is the basic program production rule. It requires the concatenation of the symbols R and M, which represent the role assignments and modes, respectively. Note that R has a Kleene-closure operator, denoted '$\star$,' which means that our programs expect zero or more role assignments; additionally, the '+' operator requires that the program have at least one mode. The R production, which creates role assignments, takes the "identifier" nonterminal I, which is similar to a variable name in standard programming languages, followed by the helper terminal, $=$, and role map terminal, r. For example, the R productions described for the MDLn string in (29) are written as:

```
a1 = 1
a2 = 0
a3 = 0
```

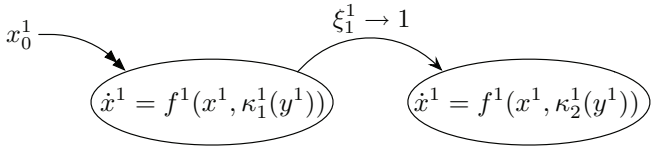
The mode production, M, is made by concatenating the identifier nonterminal I with terminals k and z, which are the controller and interrupt symbols, and a string of identifier symbols, $I^\star$. Using this production we can write the mode string corresponding to (29) as:

```
(a2, GoToGoal, Obstacle, {})
(a1, Follow, Obstacle, {a2})
(a3, Follow, Obstacle, {a1})
```

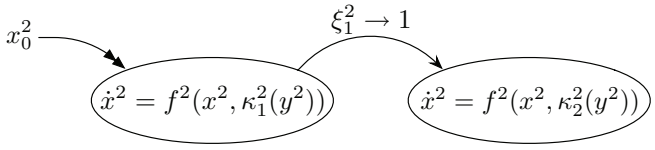
In this code snippet, the controllers from the grammar, k, are represented by the symbols `GoToGoal` and `Follow`. Additionally, the z element from the mode production is represented by the three `Obstacle` symbols. The buddy lists for each agent are the identifiers listed between the `{}` elements. This grammar allows us to create MDLn programs that are *syntactically* correct; however, we still need to design a process that ensures that MDLn programs can execute correctly when deployed onto the networked agents.

3.2 MDLn Program Consistency

Before deploying an MDLn program onto a team of agents, we must check whether the program correctly specifies the network information dependencies of the agents, which are represented by the buddy lists within each MDLn mode. We develop this consistency checker based on techniques from discrete event systems (DES) and language theory (e.g. [17, 29, 51]). First, we need to model the execution of each agent's MDLn string, by constructing an *MDL string automaton* that represents the sequential execution of the system. For example, consider the following MDLn string: $\sigma_1^1\sigma_2^1\sigma_1^2\sigma_2^2$. The execution of this string on agent-1 and agent-2 is best illustrated by the hybrid dynamics in Figure 18. Additionally, Figure 19 shows the automata corresponding to agent-1 and agent-2 executing the dynamics of the hybrid automata of Figure 18.



(a) Dynamics of agent-1.

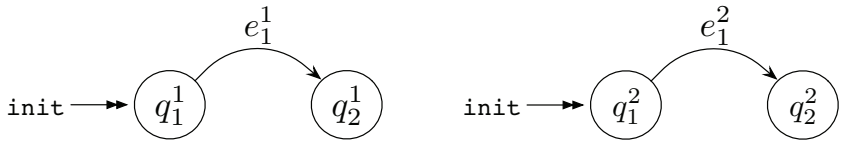


(b) Dynamics of agent-2.

Figure 18: The hybrid automata representing the dynamics of the two robots as they execute their given MDLn strings.

Automaton, A^1, in Figure 19(a) starts out running in state q_1^1, which has the output map $o(q_1^1) = \beta_1^1$. While in the state q_1^1, the dynamics of the system are taken from the first state in Figure 18(a): $\dot{x}^1 = f^1(x^1, \kappa_1^1(y^1))$. Once this mode is interrupted by $\xi_1^1 \to 1$, the dynamics are changed to $\dot{x}^1 = f^1(x^1, \kappa_2^1(y^1))$ and the event e_1^1 causes

a transition in A^1 to its next state, q_2^1. The automaton, A^2, executes in a similar manner; however, its controllers and interrupts are independent of A^1's. A formal definition of the automata in Figure 19 follows:



(a) A^1 (b) A^2

Figure 19: Two string automata representing the MDLn strings for agent-1, 19(a), and agent-2, 19(b).

Definition 3.6 (MDL String Automaton)**.** An *MDL string automaton* is the tuple

$$A^i = (Q^i, E^i, \delta, q_0^i, o^i),$$

where:

- Q^i is the set of states q_k^i and each state corresponds to agent-i executing the controller κ_k^i
- E^i is the set of events e_k^i and each event is associated with agent-i transitioning to another controller via $\xi_k^i \rightarrow 1$
- $\delta : Q^i \times E^i \rightarrow Q^i$ is the transition function, i.e. $\delta(q_k^i, e_k^i) = q_\ell^i$ means that there exists an transition, e_k^i, from q_k^i to q_ℓ^i
- q_0^i is the initial state.
- $o^i : Q^i \rightarrow 2^{\mathcal{N}}$ is the output function that maps each state to the buddy list of that current MDLn mode, i.e. for some q_k^i, $o(q_k^i) = \beta_k^i \in 2^{\mathcal{N}}$, where $\mathcal{N}$ is the agent index set

These string automata adequately represent the *individual* behavior of each agent; however, the MDLn language is designed to specify tasks for a collection of agents. Therefore, we use these automata to analyze how the networked information dependencies of each agent affect the execution of the MDLn program. A natural operation for analyzing these automata is the *parallel composition*, which shows how the execution of each agent's string affects all of the agents' behavior during execution of their individual MDLn strings. These composed automata are the subject of the next section.

3.2.1 Network Automata

Consider, again, the automata in Figure 19. Using parallel composition, we generate the automaton shown in Figure 20, and denote it as $\mathcal{A}^{12} = A^1 \| A^2$. This automaton has four states and two possible events, which are taken from the event sets of the individual automata, i.e. $E = E^1 \cup E^2 = \{e_1^1, e_1^2\}$. This automaton represents the global behavior of the agents as each executes its own MDLn string. This automaton is a key element for the analysis of MDLn programs to determine if the information dependencies of each agent are consistent during the execution of their motion programs.

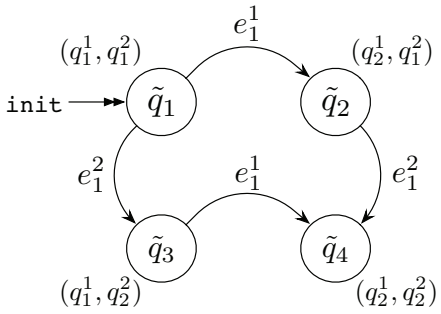


Figure 20: The automaton, $\mathcal{A}^{12}$, generated by the composition of the automata in Figure 19. The states from automata A^1 and A^2 are written next to the states of $\mathcal{A}^{12}$ to explicitly show the states included in the network automaton.

Assume we have n agents and each agent has a finite number of modes. An MDLn motion program is specified for all agents and this program is parsed and reorganized into a set of string automata. These automata are then combined using parallel composition into a network automaton, $\mathcal{A} = A^1 \| \cdots \| A^n$, which is defined formally as:

Definition 3.7 (Network Automaton)**.** A *network automaton* is defined by the tuple:

$$\mathcal{A} = (\mathcal{Q}, \mathcal{E}, \tilde{\delta}, \tilde{q}_0, \mathcal{Q}_m).$$

where:

- $\mathcal{Q}$ is the set of states such that $\tilde{q} \in \mathcal{Q} = Q^1 \times \cdots \times Q^n$, where $Q^i, i = 1, \cdots, n$ are the states from agent-i's MDL string automaton
- $\mathcal{E}$ is the set of events resulting from the union of the individual event sets from each agent, i.e. $\mathcal{E} = E^1 \cup \cdots \cup E^n$
- $\tilde{\delta} : \mathcal{Q} \times \mathcal{E} \to \mathcal{Q}$ is the transition function, i.e. $\tilde{\delta}(\tilde{q}_k, e^i_k) = \tilde{q}_\ell$ means that there exists an transition, e^i_k, from $\tilde{q}_k$ to $\tilde{q}_\ell$
- q_0 is the initial state
- $\mathcal{Q}_m$ is the set of marked states.

Network automata are used to determine whether an MDLn program is inconsistent. To do so, any state in the network automaton with incorrect information dependencies should be *marked*. Then, the goal of the consistency checker is to determine whether there are languages over the automaton that lead to marked states. The definition and identification of these inconsistent states is discussed in the following section.

3.2.2 Inconsistent States

The consistency of a state in the network automaton is determined by the buddy lists in each agent's MDLn string as well as their roles. Returning to our two agent example, if we assign roles to both agents then we change the network information dependencies of the MDLn program. In this two-agent case, we construct a *pairwise* logic predicate that indicates if a state from the two-agent network automaton in Figure 20 should be marked:

$$(q^1_{k_1}, q^2_{k_2}) \in \mathcal{Q}_m \Leftrightarrow P(q^1_{k_1}, q^2_{k_2}) \tag{31}$$

where,

$$\begin{aligned} & P(q^1_{k_1}, q^2_{k_2}) := \\ & (a^1 \notin o(q^2_{k_2}) \wedge a^2 \in o(q^1_{k_1}) \wedge r(a^1) < r(a^2)) \\ \vee \quad & (a^2 \notin o(q^2_{k_2}) \wedge a^1 \in o(q^2_{k_2}) \wedge r(a^2) < r(a^2)), \end{aligned} \tag{32}$$

where $\vee$ and $\wedge$ denote logical disjunction and conjunction, respectively. The logical statement expressed by equation (31) means that a state from $\mathcal{A}^{12}$ is inconsistent if and only if a^1 is *not* in a^2's buddy list *and* a^1 depends on a^2 *and* a^1's role value is less than a^2's value; or, *vice versa*.

To generalize equation (31) to the case of n-agents, we take the disjunction over all possible pairs that are part of state, $\tilde{q}_\ell \in \mathcal{Q}$. (If the pair $(q^i_{k_i}, q^j_{k_j})$ is part of the state $\tilde{q}_\ell$, we denote the relationship by the symbol '$\subset$'.) In other words, a state in $\mathcal{Q}$ is inconsistent if the following definition holds:

Definition 3.8 (Inconsistent State)**.** A state $\tilde{q}_\ell \in \mathcal{Q}$ is *inconsistent* if it is marked using the following relation:

$$\tilde{q}_\ell \in \mathcal{Q}_m \Leftrightarrow \bigvee_{(q^i_{k_i}, q^j_{k_j}) \subset \tilde{q}_\ell} P(q^i_{k_i}, q^j_{k_j}), \tag{33}$$

where k_i denotes the k^{th} mode of a^i.

Returning to the two-agent example network automaton in Figure 20, assume that the particular MDLn program sets the role values as $r(a^1) < r(a^2)$. Additionally, let a^1 depend on a^2 while executing its second mode, i.e. $o^1(q_2^1) = \{a^2\}$; however, a^2 operates independently while executing its first mode: $o^2(q_1^2) = \emptyset$. Applying equation (33) to each state in $\mathcal{Q}$ results in the marking of $\tilde{q}_2$, shown in Figure 21, since $a^1 \notin o^2(q_1^2) = \emptyset$, $a^2 \in o^1(q_2^1)$, and a^1's role value is lower than a^2's value.

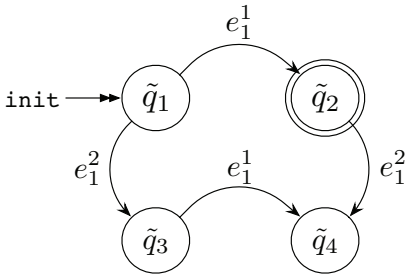


Figure 21: The two-agent network automaton, $\mathcal{A}^{12}$, with state $\tilde{q}_2$ marked as inconsistent.

An automaton with marked states generates a *marked* language, or set of strings, that enumerates the behaviors of the system leading to inconsistent states. The marked language of the network automaton in Figure 21 is $\mathcal{L}_m(\mathcal{A}^{12}) = \{e_1^1\}$, since e_1^1 is the only event that leads to $\tilde{q}_2$. This framework facilitates the identification of inconsistent MDLn motion programs; however, we would like to prevent the execution of inconsistent states to prevent errors while the agents run their MDLn programs. In the following section we will see how this marked language can be used to develop a supervisor that disables events leading to inconsistent states, such as $\tilde{q}_2$.

3.3 Supervisors for Multi-Agent Motion Programs

Section 3.2 developed a way to model an MDLn program specification and indicate when an MDLn program will not execute correctly because of inconsistent states. In order to prevent the transition into inconsistent states of $\mathcal{A}$, we propose a *supervisor*

that accepts events from the network automaton, $\mathcal{A}$, and, furthermore, blocks events in $\mathcal{A}$ that lead to inconsistent states. This supervisor is best visualized as block $\mathcal{S}$ in Figure 22, where we see that it accepts event strings from $\mathcal{A}$ and outputs some control signal, γ.

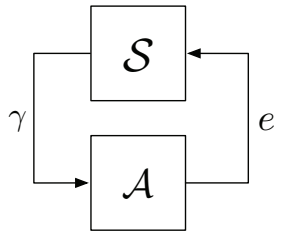


Figure 22: An illustration of a supervisor $\mathcal{S}$ monitoring the event strings, e, generated by $\mathcal{A}$. The supervisor applies an output mapping $\phi(w)$ on the current state of $\mathcal{S}$, to alter the behavior of $\mathcal{A}$.

These controls are symbols from a subset of the possible events in the network automaton: $\gamma \in \Gamma \subseteq \mathcal{E}$. We define the mapping $\phi : W \rightarrow \Gamma$ that maps the currently active supervisor state to a control input, $\gamma \in \Gamma$. The network automaton is given the controlled subset of events, which in turn disables the events leading to inconsistent states.

More formally, we state the following definition of an MDLn supervisor:

Definition 3.9 (MDLn Supervisor)**.** An *MDLn supervisor* is an automaton represented by the tuple

$$\mathcal{S} = (W, \mathcal{E}, \lambda, w_0, \phi)$$

where:

- W is the set of states
- $\mathcal{E}$ is the set of events from $\mathcal{A}$
- $\lambda : W \times \mathcal{E} \rightarrow W$ is the transition function among states in W

- w_0 is the initial state
- $\phi : W \rightarrow \Gamma$ is a function that outputs the currently enabled events at a state $w \in W$.

Note that the automaton $\mathcal{A}^{12}$ in Figure 21 will enter into the marked state $\tilde{q}_2$ if the event e_1^1 is taken from state $\tilde{q}_1$. To prevent this behavior from occurring, we construct $\mathcal{S}$ using Algorithm 3.1, which explores the network automaton with a depth-first-search (DFS) [18] adding states to the supervisor and augmenting the function ϕ at each state.

Algorithm 3.1 MDLn Supervisor Construction. This algorithm uses depth-first search to visit the states within the given network automaton, $\mathcal{A}$. The algorithm returns the constructed supervisor, $\mathcal{S}$, upon completion.

`build_supervisor` **procedure:**
Require: $\mathcal{A}$
  $W = \emptyset$ {Initialize supervisor state set.}
  $s_{curr} = \emptyset$ {Event string.}
  $\tilde{q}_{curr} = \tilde{q}_0$, $e_{curr} = \emptyset$
  `visit`($\tilde{q}_{curr}$, e_{curr}) {Perform visit to current state, $\tilde{q}_{curr}$}
  return $\mathcal{S}$

`visit` **procedure:**
Require: $\tilde{q}_{curr}$, e_{curr}
  $s_{curr} = s_{curr} \cdot e_{curr}$ {Append current event to current event string.}
  if $\tilde{q}_{curr} \notin \mathcal{Q}_m$ **then**
    w_{curr} = `new_state` {Create new supervisor state.}
    $\phi(w_{curr}) = \emptyset$
    for all e leaving $\tilde{q}_{curr}$ **do**
      if $(s_{curr} \cdot e) \notin \mathcal{L}_m(\mathcal{A})$ **then**
        $\phi(w_{curr}) \cup e$
      end if
    end for
    $W \leftarrow W \cup w_{curr}$
    $\tilde{q}_{adj}$ = `adjacent`($\tilde{q}_{curr}$) {Get adjacent states to $\tilde{q}_{curr}$.}
    for all $\tilde{q}_{adj}$ **do**
      `visit`($\tilde{q}_{adj}$, e_{adj}) {Perform recursive depth-first search call.}
    end for
  end if

As an example of the execution of Algorithm 3.1, the supervisor for $\mathcal{A}^{12}$ in Figure

21 is generated in the following way. We start at $\tilde{q}_1$ in $\mathcal{A}^{12}$ and create an initial state in W, $w_0 := w_1$, as shown in Figure 23. At this state, the current event string is simply the empty string, ϵ, and the control function is initialized as $\phi(w_1) = \emptyset$. As we iterate over the possible transitions out of $\tilde{q}_1$, only $e_1^2 \notin \mathcal{L}_m(\mathcal{A}^{12})$; therefore, $\phi(w_1) = \{e_1^2\}$. Since e_1^1 leads to an inconsistent state, we do not traverse beyond that transition. Next, we return to state $\tilde{q}_1$ (via DFS) and take the last remaining event to $\tilde{q}_3$, which is a consistent state. We create the next state, w_2, in the supervisor and connect it to w_1 with event e_1^2. The enabling function at w_2 is set to $\phi(w_2) = \{e_1^1\}$ since $e_1^2 \cdot e_1^1 \notin \mathcal{L}_m(\mathcal{A}^{12})$. After making the final jump to state $\tilde{q}_4$ with e_1^1, we see that there are no more states in $\mathcal{Q}$ and $\tilde{q}_4$ is not an inconsistent state; therefore, the process completes by adding with a final state w_3 to the supervisor and setting $\phi(w_3) = \emptyset$.

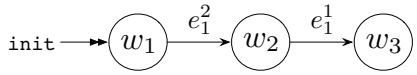


Figure 23: The constructed supervisor for the network automaton $\mathcal{A}^{12}$ in Figure 21.

3.3.1 MDLn Supervisor Deployment

In summary, the MDLn compilation process involves the inconsistency analysis of Section 3.2 and the *automatic* generation of a supervisor automaton. If a MDLn program is created such that its initial state is inconsistent, then a supervisor automaton will not be generated. Otherwise, the supervisor automaton is created and is deployed within a *supervisor agent* that can monitor the other MDLn agents on the network.

This supervisor agent inspects the current state of its automaton and applies the $\phi(\cdot)$ function. If any events should be disabled on the network, the supervisor issues `hold` messages to those agents with the disabled events. Additionally, as the MDLn agents execute their programs, they transmit `transition` messages that cause

the supervisor to advance its automaton and setup the next set of enabled events. Examples of this implementation are described in the next section.

3.4 Simulation Results

In this section, we demonstrate the simulation of the MDLn framework discussed through Sections 3.1–3.3 using the robotics simulation environment *Player/Stage* [27] as our back-end. We use our multi-agent software infrastructure, *Pancakes*, which will be discussed further in Chapter 5, to create and manage the agents on the network.

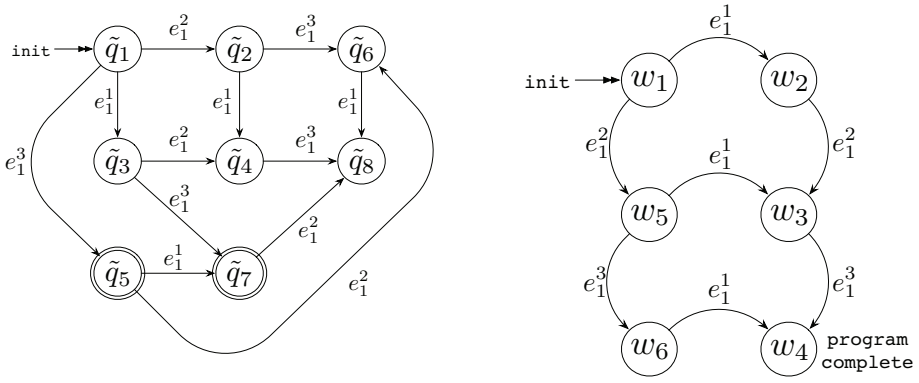
3.4.1 Example: Basic Supervisor Operation

Assume we are given two agents that share a leader role and a single follower, i.e. $r(a^1) = r(a^2) > r(a^3)$. The MDLn program is:

```
a1 = 1, a2 = 0, a3 = 0
(a1, GoToGoal, AtGoal, {a3})
(a1, Wait, Always, {a3})
(a2, GoToGoal, AtGoal, {})
(a2, Wait, Always, {a3})
(a3, Follow, BuddyAtGoal, {a1})
(a3, Follow, Obstacle, {a2})
```

This MDLn program instructs a^1 to approach the goal and then, once the goal is reached, wait indefinitely; additionally, a^1 is instructed to share its information with a^3. A similar string is given to a^2, but it excludes a^3 from knowing its information until it switches to its second mode: `(a2, Wait, Always, {a3})`. Finally, a^3 is instructed to follow a^1 until a^1 reaches the goal, and then switch to following a^2 until it detects an obstacle. Note that a^3 uses the `BuddyAtGoal` interrupt for determining when it should switch to following a^2. Since `BuddyAtGoal` uses the shared (and possibly noisy)

information of a^1, it is possible for the interrupt to fire before a^1 actually reaches the goal.



(a) The network automaton generated by the example MDLn program.

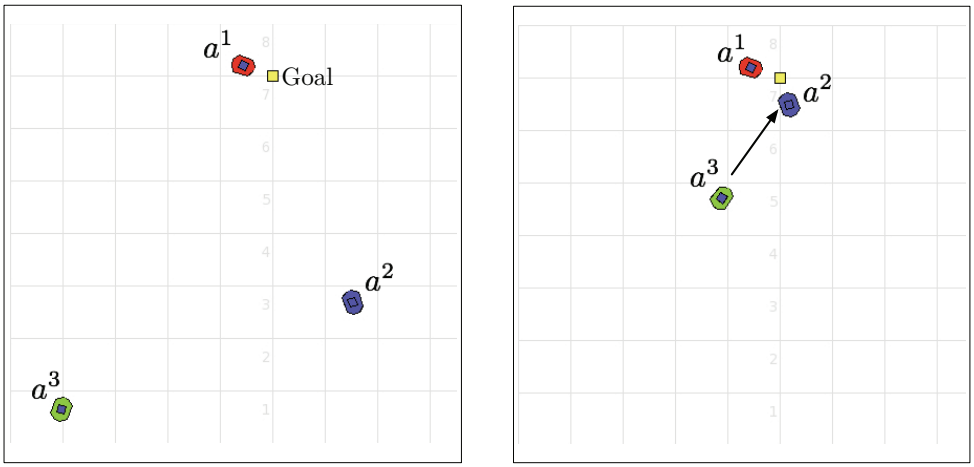
(b) The supervisor generated from the follow-the-leader MDLn program.

Figure 24: The network and supervisor automata generated by the compilation of the MDLn example program.

The network automaton generated by the compilation of this program is shown in Figure 24(a). Note that if a^3 switches to its second mode it creates an inconsistent state since it depends on the information from a^2 to execute the `Follow` controller. Additionally, the network dependencies are still inconsistent if a^1 switches to its second mode and a^3 still attempts to follow a^2. Therefore, by applying the logic predicate from equation (33), states $\tilde{q}_5$ and $\tilde{q}_7$ are marked as inconsistent.

The compilation automatically generates the supervisor shown in Figure 24(b). This supervisor automaton is deployed inside the supervisor agent, denoted `SA`, which receives messages from the MDLn agents, a^1, a^2, and a^3, every time a transition in the MDLn program occurs. When the program starts, the supervisor starts in state w_1 with only two enabled events e_1^1, e_1^2. The `SA` issues a `hold` message to a^3, which prevents the a^3 from executing its second mode (Figure 25(a)). Once a^1 reaches the goal, the event e_1^1 is transmitted to the `SA` and the supervisor advances to state w_2. In this state, a^3 is still held from advancing its program. After a^2 reaches the goal,

the event e_1^2 advances the supervisor to state w_3 and a `release` message is sent to a^3. Finally, a^3 executes its `Follow` behavior and completes the program once it reaches a^2, as shown in Figure 25(b).



(a) (b)

Figure 25: The image in 25(a) shows a^3 being held after initially following a^1. Once a^2 reaches the goal, as shown in 25(b), the `SA` releases a^3 and a^3 follows a^2 to the goal.

3.4.2 Example: Threat Detection

In this example, we construct a more complicated MDLn motion program that uses four heterogeneous agents. We partition the network using the following role values:

$$a^1 = 2, a^2 = a^3 = 1, a^4 = 0.$$

The a^1 agent has a sensor that can identify possible threats to the other agents. Two other agents, a^2 and a^3, are tasked with exploring the environment and a^4 is set to follow a^3 for exploration redundancy.

The MDLn program scripted for this example is:

```
(a1, GoToGoal, ThreatDetected, {a2 a3})
(a1, ApproachThreat, AtThreat, {})
```

```
(a1, ScanThreat, 10, {})
(a1, GoToGoal, AtGoal, {a2 a3})
(a2, Explore, Obstacle or 10, {})
(a2, Follow, Obstacle, {a1})
(a2, Avoid, Clear, {})
(a3, Explore, Obstacle or 10, {a4})
(a3, Follow, Obstacle, {a1})
(a3, Avoid, Clear, {})
(a4, Follow, Obstacle, {a3})
(a4, Avoid, Clear, {})
(a4, GoToGoal, AtGoal, {})
```

This more complicated program has a^1 go to the goal until it detects a threat, all the while sharing information with a^2 and a^3. When it detects the threat, it must approach and perform a `ScanThreat` behavior for 10 time units. After threat scanning is complete, a^1 is instructed to continue towards the goal. The strings for a^2 and a^3 are similar: they both explore until they see an obstacle *or* 10 time units have passed. Once that is complete they both follow a^1 until an obstacle is detected. Finally, a^4 follows a^3, which has $a^4 \in \beta^3$, until a^4 detects an obstacle. If that event occurs, a^4 continues to the goal position.

The network automata generated by this program has 108 possible states, 46 of which are inconsistent, and the resulting supervisor contains 62 states. We deploy this program onto the four agents shown in Figure 26 and a similar supervisor agent, `SA`. This example program shows how much more complicated the network automata and supervisors become by adding a few modes and agents. However, it also shows that our automatic tool is useful for the analysis of the potentially complicated network interactions specified by an MDLn program.

Initially, a^1 detects and approaches the target in Figure 27(a). At this point in

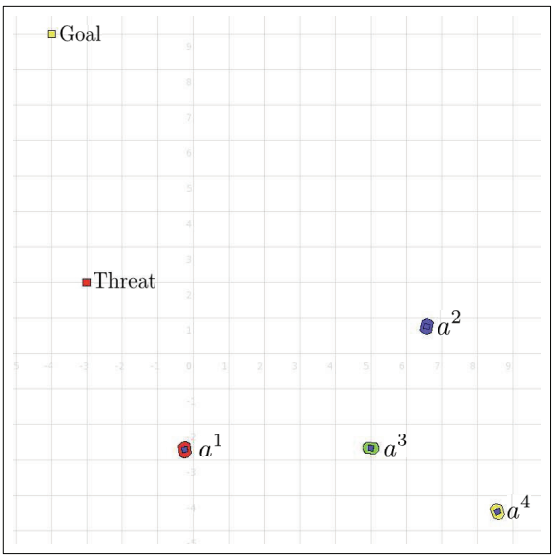
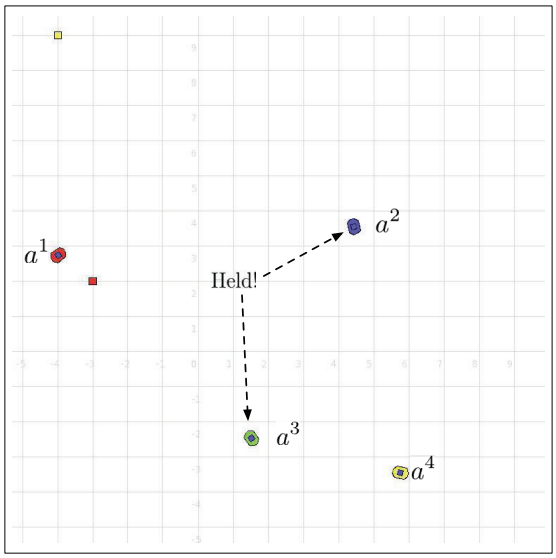


Figure 26: The initial configuration of the four agents for the threat detection example.

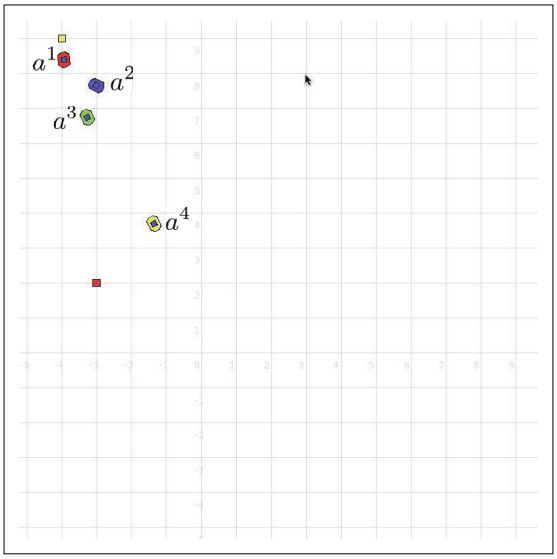
the program, a^2 and a^3 cannot receive a^1's position information for following, since a^1 is scanning the threat by itself. The `SA` holds a^2 and a^3 to prevent the entry into an inconsistent state; however, a^4 can still follow a^3. Once a^1 finishes scanning the threat, it proceeds towards the goal and again a^2 and a^3 are allowed to get a^1's shared information. The event e_3^1 of a^1's MDLn string fires and advances the supervisor automaton in `SA`, which re-enables a^2 and a^3. Figure 27(b) shows the completion of the program, as a^1 reaches the goal with a^2 and a^3 following and a^4 executing its `GoToGoal` behavior.

3.5 Conclusions

In this chapter, we designed a networked MDL, or MDLn, that enables the specification of motion programs for multi-agent systems. We developed a framework for identifying "bad" MDLn states by constructing a network automaton, which enumerate the global behavior of the specified program, and marking states of the automaton if they violate the agent interaction rules of MDLn. Furthermore, we created a



(a)



(b)

Figure 27: Images of the simulation running the example MDLn program. The image in Figure 27(a) shows a^2 and a^3 being held by the `SA` while a^1 scans the target. Figure 27(b) shows the final completion of the simulation with all agents approaching the final destination.

depth-first-search based algorithm that automatically constructs an MDLn supervisor. This supervisor entity operates in parallel with the agents as they execute their

MDLn programs and prevents agents from entering into the marked states of the network automaton. We demonstrated the effectiveness of the MDLn framework on two different teams of simulated, robotic agents.

CHAPTER IV

COMPILING MOTION PROGRAMS WITH BOUNDED INPUTS

So far we have considered the problem of turning MDL specifications into executable code on systems with temporal, energy, and spatial constraints. Usually, these constraints are encoded within cost functionals that serve as the basis for an MDL compiler. When specifying such motion programs, it is generally assumed that the controller "fits" the dynamics of the system in that the system can execute the control string. However, in a number of practical applications, e.g. embedded control systems, there are hard constraints on the actuator signals achievable that effect what motion programs can be executed. Therefore, it is possible that a motion program that is intended to perform some action, actually fails to accomplish the task because of the input constraints.

In this chapter, we consider the case where our target system may not be able to execute the specified motion program due to actuator limits. Our goals are to identify if a motion program is feasible and, if not, insert new MDL modes that can make the motion program execute correctly. These research results are related to recent work seen in [10, 34, 48, 59]. In [10], the authors developed an algorithm for controlling robotic manipulators in constrained environments using linked, invariant sets. The work in [34] devised a supervisory control algorithm that selects a controller from a set to achieve performance while balancing stability requirements under input and state constraints. Finally, the work in [48, 59] is more closely related to our approach, since they use a Lyapunov-based scheduling procedure to select gains for controllers to maintain system stability.

Our method for compiling motion programs with bounded inputs differs from this prior work by focusing on minimizing the number of mode insertions, while maintaining the bounded input constraints. In Section 4.1, we develop the controllers and associated MDL for controlling a linear system that has bounded input signals. Section 4.2 derives the regions of attraction for controllers that drive the system to particular set-points. Furthermore, in Section 4.3, we construct an algorithm, that compiles motion programs while considering the input signal bounds. Finally, the simulation results in Section 4.4 demonstrate the effectiveness of the MDL compiler.

4.1 Motion Programs for Moving Between Set-Points

To develop our initial compilation framework, we restrict the MDL to take on a particularly simple form, in that we will insist on the control laws being affine in the state, driving the system to particular equilibrium points, and the interrupts triggering when these points are reached. Consider the unstable, controllable linear system

$$\dot{x} = Ax + bu,\ x \in \mathbb{R}^n,\ u \in \mathbb{R}. \tag{34}$$

where $x \in \mathbb{R}^n$ and $u \in \mathbb{R}$, and A, b are matrices of appropriate dimension. A stabilizing controller for this system can be generated by solving the LQ design problem where the cost functional is

$$J(x,u) = \int_0^\infty \left(x^T Q x + \sqrt{2} u^T u \right) dt.$$

The solution is given by the feedback gain P that solves the Riccati equation

$$A^T P + PA + Q - 2Pbb^T P = 0, \tag{35}$$

where $Q \succ 0$. The resulting feedback control for stabilizing (34) is thus given by

$$u = -b^T P x. \tag{36}$$

We select such a P and then augment the controller with an affine term that drives the system to a given set-point. By stringing together these different affine terms,

we get a motion programs that takes the system through a sequence of set-points. Consequently, we modify the control input to include an affine term, v, as

$$u = -b^T P x + v, \tag{37}$$

which results in the very standard control architecture seen in Figure 28.

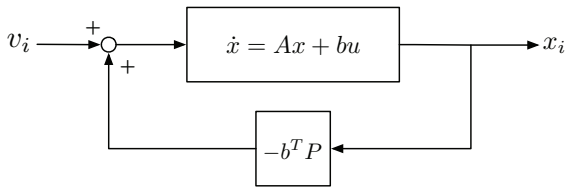


Figure 28: The basic control architecture for feeding MDL elements to the system.

By applying the controller (37) to (34), we get the closed loop dynamics:

$$\dot{x} = (A - bb^T P)x + bv,$$

with globally attractive, stationary points given by

$$x_v = -(A - bb^T P)^{-1} bv.$$

The inverse, $(A - bb^T P)^{-1}$, is well defined since the real parts of its eigenvalues are negative by design. Therefore, we can define the interrupt function $\xi(P, v, \epsilon)$ that triggers once the state of the system is close to the desired set-point x_v, i.e.

$$\xi(P, v, \epsilon) = \begin{cases} 1 & \text{if } \|x - x_v\|_P^2 \leq \epsilon \\ 0 & \text{otherwise,} \end{cases} \tag{38}$$

where $\|z\|_P^2 = z^T P z$.

Consequently, the motion program used for driving the system through a collection of set-points is given by strings of the form $((P, v_1), \xi(P, v_1, \epsilon_1)), \ldots, ((P, v_N), \xi(P, v_N, \epsilon_N))$. Since we assume that P is fixed, we can use the shorthand $(v_1, \epsilon_1), \ldots, (v_N, \epsilon_N)$ to denote these MDL strings.

4.2 Compiling Motion Programs under Input Constraints

From our development in Section 4.1, we have a controller for each $v \in \mathbb{R}$ that takes the system (asymptotically) to the set-point x_v. The first thing we need to do in order to "compile" the MDL programs is to characterize what the set of such set-points actually looks like under the input constraint $|u| < u_{max}$.

4.2.1 Regions of Attraction

Let the set of stationary points be denoted by $\mathcal{X}$. In order to calculate the regions of attraction around each point, we need the following lemma:

Lemma 4.1 (Stationary Points under Bounder Input)**.** *Let (A, b) be a controllable pair, let P be the positive definite matrix solution to the Riccati equation* (35) *for some $Q \succ 0$, and let $K = (A - bb^T P)$. If $u = -b^T Px + v$ and $|u| < u_{max}$, then the set of stationary points $\mathcal{X}$ is given by*

$$\mathcal{X} = -K^{-1}b(b^T PK^{-1}b + 1)^{-1}[-u_{max}, u_{max}]$$

if $b^T PK^{-1}b + 1 \neq 0$ or,

$$\mathcal{X} = -K^{-1}b\mathbb{R}, \text{ otherwise.}$$

Proof 4.1. Assume that $x_{v_i} \in \mathcal{X}$ is the stationary point obtained by using the open-loop control v_i in equation (37). Then, the total control effort needed to hold the system at x_{v_i} is

$$\begin{aligned} u_{v_i} &= -b^T Px_{v_i} + v_i \\ &= (b^T PK^{-1}b + 1)v_i \end{aligned}$$

Note that if $b^T PK^{-1}b + 1 = 0$ then $u_{v_i} = 0$ for any $v \in \mathbb{R}$. If the equality *does not* hold, then the choice of v must come from the set

$$v \in \mathcal{V} = (b^T PK^{-1}b + 1)^{-1}[-u_{max}, u_{max}].$$

in order to maintain that $|u| < u_{max}$; otherwise, $v \in \mathbb{R}$. Hence, the set of stationary points is

$$\mathcal{X} = -K^{-1}b\mathcal{V}$$

which completes the proof. □

Now, with a proper characterization of these stationary points, we can proceed with calculating the regions of the state space from which these points can be reached with bounded inputs. These regions will form the basis for developments in the following sections.

Theorem 4.1 (Regions of Attraction Under Bounded Input)**.** *Given the assumptions in Lemma 4.1, the region of attraction around the point x_{v_i} is given by*

$$\mathcal{E}(P, v_i) = \{x \in \mathbb{R}^n | (x - x_{v_i})^T P (x - x_{v_i}) \leqslant \alpha_{v_i}\} \tag{39}$$

with

$$x_{v_i} = -K^{-1}bv_i$$

and

$$\alpha_{v_i} = (b^T P b)^{-1}(u_{max} - |u_{v_i}|)^2. \tag{40}$$

Proof 4.2. Let $x = x_{v_i} + \Delta x_{v_i}$, then

$$u = -b^T P(x_{v_i} + \Delta x_{v_i}) = u_{v_i} - b^T P \Delta x_{v_i}.$$

However, since we have the constraint $u \in [-u_{max}, u_{max}]$, we interpret the above equation as

$$b^T P \Delta x_{v_i} \in [-u_{max} + u_{v_i}, u_{max} + u_{v_i}].$$

P is a solution to the Riccati equation (35), so we define the function

$$V(\Delta x_{v_i}) = \Delta x_{v_i}^T P \Delta x_{v_i},\ \forall \Delta x_{v_i} \in \mathbb{R}^n,\ \Delta x_{v_i} \neq 0. \tag{41}$$

If this function is Lyapunov, then it can serve as a conservative region of attraction around the point x_{v_i}. To show this fact, consider the ellipsoid generated by $\Delta x_{v_i}^T P \Delta x_{v_i} = \gamma$, where $\gamma > 0$ and real and $\forall \Delta x_{v_i} \in \mathbb{R}^n$. Assume γ is chosen such that $| - b^T P \Delta x_{v_i}| < u_{max}$, and, furthermore, we choose some $\beta \in (0, \gamma)$.

Thus, we want to solve the following maximization problem for any $\Delta x_{v_i} \in \mathbb{R}^n$:

$$\max_{\Delta x_{v_i}} b^T P \Delta x_{v_i}$$
$$\text{s.t.} \quad \Delta x_{v_i}^T P \Delta x_{v_i} = \gamma.$$

Forming the Lagrangian,

$$L = b^T P \Delta x_{v_i} - \lambda(\Delta x_{v_i}^T P \Delta x_{v_i} - \gamma),$$

and setting its derivative w.r.t. Δx_{v_i} equal to 0 we get that

$$\Delta x_{v_i} = -\frac{1}{2\lambda} b \tag{42}$$

which implies that the maximum Δx_{v_i} is parallel to the input matrix b. According to the constraint equation we calculate the value of λ and insert into equation (42), resulting in the maximum value:

$$\Delta x_{max} = \sqrt{\frac{\gamma}{b^T P b}} b.$$

Using this value in the control law results in the maximum control effort

$$u_\gamma = b^T P \sqrt{\frac{\gamma}{b^T P b}} b = \sqrt{\gamma} \|b\|_P$$

Finally, if we compare the difference between the maximum input along the the γ level-set and that on the β level-set we get

$$u_\beta - u_\gamma = (\sqrt{\beta} - \sqrt{\gamma}) \|b\|_P$$

which is a strictly negative quantity, by the assumption that $\beta \in (0, \gamma)$. Therefore, the control effort reduces as the state decays within the ellipsoid $\Delta x_{v_i}^T P \Delta x_{v_i}$, and (41) is a Lyapunov equation.

Now that we know the region defined by (41) is Lyapunov, we find the solution to

$$\min_{\Delta x_{v_i}} \Delta x_{v_i}^T P x_{v_i}$$

such that

$$b^T P \Delta x_{v_i} = -u_{max} + u_{v_i}$$

or

$$b^T P \Delta x_{v_i} = u_{max} + u_{v_i}.$$

Letting $c = Pb$ and $d_\pm = \pm u_{max} + u_{v_i}$, the Lagrange necessary and sufficient conditions (see for example [37]) for these quadratic optimization problems are:

$$\begin{aligned} P\Delta x_{v_i} + c\lambda_{v_i} &= 0 \\ c^T \Delta x_{v_i} - d_\pm &= 0 \end{aligned}$$

where $\lambda_{v_i} \in \mathbb{R}^n$ is the Lagrange multiplier. Solving these equations results in

$$\begin{aligned} \lambda_{v_i} &= -(c^T P^{-1} c)^{-1} d_\pm \\ \Delta x_{v_i} &= P^{-1} c (c^T P^{-1} c)^{-1} d_\pm. \end{aligned}$$

Inserting the solution for Δx_{v_i} into (41) results in:

$$\Delta x_{v_i}^T P \Delta x_{v_i} = (b^T P b)^{-1} (\pm u_{max} + u_{v_i})^2$$

where $(b^T P b)^{-1}$ exists since $P \succ 0$ and $b \neq 0$ by our controllability assumption. We choose the smallest and define it as

$$\alpha_{v_i} = (b^T P b)^{-1} (u_{max} - |u_{v_i}|)^2.$$

Therefore, the region around each stationary point x_{v_i} where $|u| \leq u_{max}$ is given by

$$\mathcal{E}(P, v) = \{x \in \mathbb{R}^n | (x - x_{v_i})^T P (x - x_{v_i}) \leq \alpha_{v_i}\},$$

as in equation (39). □

A corollary of Theorem 4.1 describes the characterization of the entire region of attraction about the points in $\mathcal{X}$:

Corollary 4.1. *The region of attraction around the set of stationary points $\mathcal{X}$ is given by*

$$\mathcal{L} = \bigcup_{v_i \in \mathcal{V}} \mathcal{E}(P, v_i).$$

Figure 29 shows an example of the application of Theorem 4.1 to (34) with matrices

$$A = \begin{pmatrix} 1 & 2 \\ 0 & -1 \end{pmatrix}, \; b = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \; Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

and $u_{max} = 1$. This figure shows some of the stationary points generated by Theorem 4.1 at the center of their attractive regions.

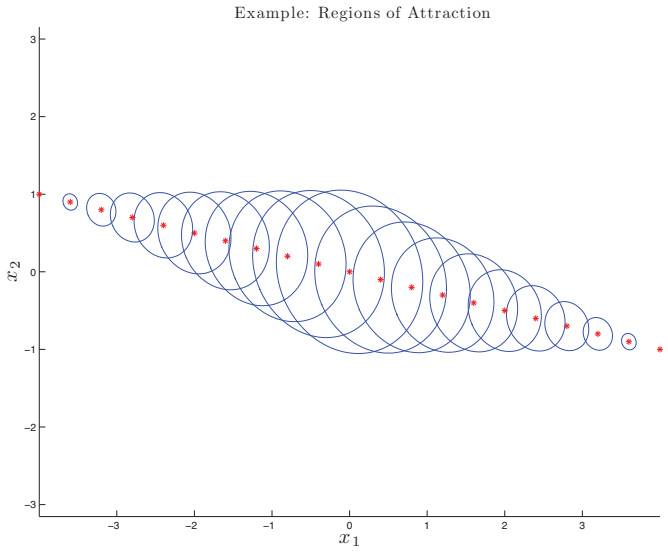


Figure 29: An example of the regions of attraction for the given example system with $u_{max} = 1$.

4.2.2 Checking for Feasibility

We are interested in determining whether a given MDL program $(v_1, \epsilon_1), \ldots, (v_N, \epsilon_N)$ will successfully drive the system between the desired set-points under the input

constraint $|u| \leq u_{max}$. It is clear that v_i must be in $\mathcal{V}$ for this to be possible, so we first make the following assumption:

Assumption 4.1. $v_i \in \mathcal{V},\ i = 1, \ldots, N.$

Now, in order for a MDL string to be feasible, the i^{th} set-point must lie in the region of attraction for the $(i+1)^{th}$ set-point, and we state this fact as a lemma:

Lemma 4.2. *Given an MDL string*

$$\sigma = (v_1, \epsilon_1), \ldots, (v_N, \epsilon_N),$$

satisfying Assumption 4.1, let

$$\mathcal{B}_{\epsilon_i}(v_i) = \{x \in \mathbb{R}^n \mid (x - x_{v_i})^T P (x - x_{v_i}) \leq \epsilon_i\}$$

represent an ellipse around point x_{v_i}. If

$$\mathcal{B}_{\epsilon_i}(v_i) \subseteq \mathcal{E}(P, v_{i+1}) \tag{43}$$

then x can be transferred from x_{v_i} to $x_{v_{i+1}}$ while satisfying the bounded input constraint.

(Note here that the set $\mathcal{B}_{\epsilon_i}(v_i)$ is exactly the set where interrupt in equation (38) takes on the value 1.)

This lemma states that if the ellipse of size ϵ_i around point x_{v_i} is strictly within the region of attraction of $x_{v_{i+1}}$, then the system will arrive at $x_{v_{i+1}}$ (asymptotically) and satisfy the input constraints. Based on this pairwise characterization of the feasibility, we can now extend this notion to entire MDL strings:

Assumption 4.2.

$$x(0) \in \mathcal{E}(P, v_1).$$

Definition 4.1. The string

$$\sigma = (v_1, \epsilon_1), \ldots, (v_N, \epsilon_N),$$

is a *feasible program string* if it satisfies Assumptions 4.1 and 4.2, and

$$\mathcal{B}_{\epsilon_i}(v_i) \subseteq \mathcal{E}(P, v_{i+1}), \text{ for } i = 1, \cdots, N-1.$$

The set of these feasible program strings is denoted by $\mathcal{F}$.

We state the following theorem (whose rather obvious proof we omit):

Theorem 4.2 (Program Feasibility)**.** *The MDL string σ drives the system close to the set-points (in the sense defined by the interrupts) if $\sigma \in \mathcal{F}$.*

This theorem gives us a feasibility check for determining if the string does in fact perform as desired. However, if the feasibility condition is violated[1], something must be done. In the next section, we discuss how to insert new control modes into the MDL string in order to respect the bounded input constraints and drive through the desired intermediary set-points.

4.3 Mode Insertion to Maintain Input Bounds

When an MDL string fails the feasibility check, we need to modify the string to ensure that the input constraints are satisfied. Our approach is to insert new modes into the string σ, so that the augmented string becomes a member of the feasible set $\mathcal{F}$. This method was inspired by *sequential composition*, as described in [16], by inserting modes when we know that the inserted region of attraction contains a subset of the region of attraction of the previous mode.

Consider, again, the MDL string

$$\sigma = (v_1, \epsilon_1), \ldots, (v_N, \epsilon_N).$$

[1]Since our Lyapunov functions are *conservative* estimates of the region of attraction around each point, it may still be possible to execute an MDL string even if $\sigma \notin \mathcal{F}$.

Each element in σ comes from a finite alphabet of modes, which we denote by $(v_i, \epsilon_i) \in \mathcal{A}$. The set of all possible concatenations of these elements is denoted by $\mathcal{A}^\star$; consequently, each MDL string comes from the set of all concatenations, i.e. $\sigma \in \mathcal{A}^\star$. We define the length operator as a mapping $\ell : \mathcal{A}^\star \to \mathbb{N}$, which accepts an MDL string and returns its number of elements. For example, if $\sigma = (v_1, \epsilon_1)(v_3, \epsilon_3)$, then $\ell(\sigma) = 2$.

Now, using our definitions, we state the mode insertion problem as:

$$\begin{aligned} &\min_{\sigma'} \ell(\sigma') \\ \text{s.t.} \quad &(v_i, \epsilon_i)\sigma'(v_{i+1}, \epsilon_{i+1}) \in \mathcal{F}. \end{aligned}$$

Thus, the problem of inserting intermediate points is characterized by minimizing the length of the string σ' such that the new string $(v_i, \epsilon_i)\sigma'(v_{i+1}, \epsilon_{i+1})$ is still in the set of feasible program strings, $\mathcal{F}$. This problem can be solved by inserting new modes, $(v_{k_l}, \epsilon_{k_l})$, such each pair of intermediate points satisfy the pairwise relation (43).

4.3.1 `MAXFORWARD` Algorithm

We develop an algorithm, called `MAXFORWARD`, that builds up the intermediate MDL string σ' using the relation in (43). This algorithm begins with the starting element of the MDL string: (v_i, ϵ_i). When the system uses this MDL mode, the state is pulled toward the point $x_{v_i} \in \mathcal{X}$ until it crosses into the interrupt region, $\mathcal{B}_{\epsilon_i}(v_i)$, shown in Figure 30. From this region we need to insert a finite number of modes that can drive our system to $(v_{i+1}, \epsilon_{i+1})$.

We want to choose a v_{k_l} such that $\mathcal{B}_{\epsilon_i}(v_i) \subseteq \mathcal{E}(P, v_{k_l})$. In other words, we need an ellipse that covers the interrupt region so that equation (43) in Lemma 4.2 holds. To find the value of v_{k_l} that results in the covering ellipse $\mathcal{E}(P, v_{k_l})$, we design an iterative algorithm that steps through possible values of $v \in \mathcal{V}$, starting at v_i. At each iteration we perform the update

$$v_{k_{j+1}} = v_{k_j} + \Delta v,$$

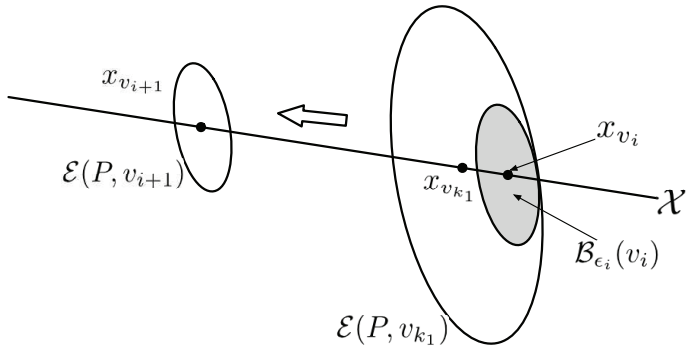


Figure 30: An illustration of the MAXFORWARD algorithm's first iteration. This iteration inserts the new mode $(v_{k_1}, \epsilon_{k_1})$ since the ellipse $\mathcal{E}(P, v_{k_1})$ covers the region of attraction around x_{v_i}.

where $\Delta v > 0$ is a fixed step size. As the size of Δv increases the more numerical error enters into the insertion process, eventually creating incorrect mode insertions. If this increment results in $\mathcal{B}_{\epsilon_i}(v_{k_j}) \nsubseteq \mathcal{E}(P, v_{k_{j+1}})$ then the value v_{k_j} is chosen as an intermediate MDL mode: $(v_{k_j}, \epsilon_{k_j})$.

The first step in this process is visualized by the ellipse $\mathcal{E}(P, v_{k_1})$ covering $\mathcal{B}_{\epsilon_i}(v_i)$ in Figure 30. We repeat the algorithm until the m^{th} step, where

$$\mathcal{B}_{\epsilon_{k_m}}(v_{k_m}) \subseteq \mathcal{E}(P, v_{i+1}).$$

At this point we have the new intermediate string:

$$\sigma' = (v_{k_1}, \epsilon_{k_1}), \ldots, (v_{k_m}, \epsilon_{k_m}),$$

which maintains that $(v_i, \epsilon_i)\sigma'(v_{i+1}, \epsilon_{i+1}) \in \mathcal{F}$. Note that this algorithm produces an optimal path given our feasibility requirements. Without input bounds, the optimal solution would be more straightforward.

The formal algorithm is shown in listing Algorithm 4.1.

Theorem 4.3 (MAXFORWARD Optimality)**.** *If Algorithm 4.1 returns a solution σ' then it produces the minimal length string σ' such that $(v_i, \epsilon_i)\sigma'(v_{i+1}, \epsilon_{i+1}) \in \mathcal{F}$.*

Algorithm 4.1 `MAXFORWARD` Algorithm

Choose $\Delta v > 0$
$v_{k_j} \leftarrow v_i$ {Initialize with first MDL mode's controller.}
`covered` $\leftarrow$ `FALSE`
while $\mathcal{B}_{\epsilon_{k_j}}(v_{k_j}) \not\subseteq \mathcal{E}(P, v_{i+1})$ **do**
  while $\neg$`covered` **do**
    $v_{k_{j+1}} = v_{k_j} + \Delta v$
    if $\mathcal{B}_{\epsilon_{k_j}}(v_{k_j}) \not\subseteq \mathcal{E}(P, v_{k_{j+1}})$ **then**
      $\sigma' \leftarrow (v_{k_j}, \epsilon_{k_j})$ {Add interim MDL mode.}
      `covered` $\leftarrow$ `TRUE`
    end if
  end while
end while
return σ'

Proof 4.3. Assume Algorithm 4.1 returned the string σ' corresponding to m intermediate points $x_{v_1}, \cdots, x_{v_m}$ between x_{v_i} and $x_{v_{i+1}}$, i.e. Algorithm 4.1 inserted m new modes into the MDL string. We note that in order to go from some x_{v_k} to $x_{v_{i+1}}$ (or more precisely, from small ellipses around these points) Algorithm 4.1 needs $m-k+1$ modes.

Now, since $v \in \mathbb{R}$, and hence $\dim(\mathbf{relint}(\mathcal{X})) = 1$, where **relint** denotes the relative interior, we can order points along $\mathcal{X}$ by how far away from $x_{v_{i+1}}$ they are. The first observation is that, by construction, the `MAXFORWARD` nature of Algorithm 4.1 prevents the existence of a point $x' \in \mathcal{X}$ such that x' can be reached in k or fewer steps from x_{v_i}, and $\|x' - x_{v_{i+1}}\| < \|x_{v_k} - x_{v_{i+1}}\|$. Instead, assume that σ' is *not* optimal and that the k^{th} intermediary point is $x'' \neq x_{v_k}$. Thus, we know that $\|x'' - x_{v_{i+1}}\| > \|x_{v_k} - x_{v_{i+1}}\|$.

But, the `MAXFORWARD` property again implies that no $x \in \mathcal{X}$ such that $\|x - x_{v_{i+1}}\| > \|x_{v_k} - x_{v_{i+1}}\|$ can reach $x_{v_{i+1}}$ in fewer steps than $m - k + 1$. As such, the optimal string (containing the intermediary point x'') can have no fewer elements than σ', and hence σ' is the (not necessarily unique) optimal solution. □

4.4 Simulation Results

For our simulation results, we use the same choice of system matrices from the end of Section 4.2.1. We specified a two mode MDL string: $(v_1, \epsilon)(v_2, \epsilon)$, where each mode uses the same sized interrupt region defined by $\mathcal{B}_\epsilon(v_i),\ i = 1, 2$ and the values of v_i come from the computed region $\mathcal{V}$.

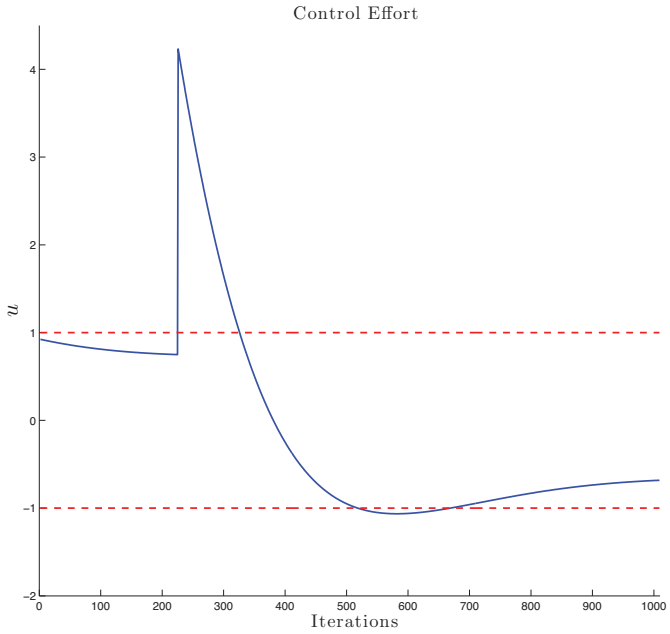


Figure 31: The plot of the input u over the iterations of the simulation. Note that at the switch point, the system requires an input far greater than what the actuators can supply. The system leaves the input bound, again, as the second mode is executed.

Figure 31 shows the effort of the feedback controller as the system executes this MDL string. Once the system reaches the interrupt region of the first mode, the system switches to (v_2, ϵ), which causes a jump in the input signal that is well outside of the upper bound of the input constraint, $u_{max} = 1$. According to Definition 4.1, this MDL string is *not* feasible; hence, we must apply Algorithm 4.1 to insert intermediate modes.

The result of our `MAXFORWARD` algorithm, with $\Delta v = 0.001$, is shown in Figure

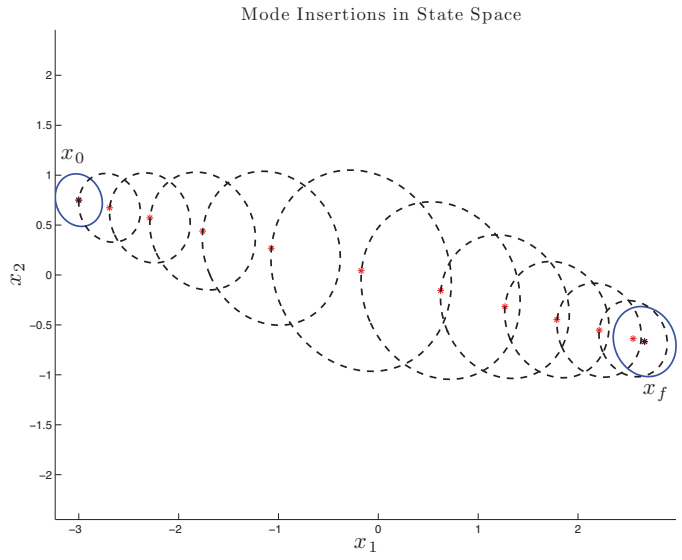


Figure 32: This figure shows the ten new ellipses (dotted lines), $\mathcal{E}(P, v_{k_j})$ for $j = 1, \cdots, 10$, produced by the intermediate modes inserted into the MDL string.

32 by the dotted ellipses. The algorithm inserted ten modes, allowing the system to move from its starting point x_0 to the final state x_f with bounded control effort, as shown in Figure 33. Using the expanded MDL string lengthens the time it takes for the system to approach x_f; however, our design goal of maintaining the input constraints was met.

4.5 Conclusions

In this chapter, we considered the problem of expanding motion programs when the system must operate under bounded input constraints. We presented the concept of a feasible MDL string and an algorithm that modifies an infeasible string so that it can execute on a system without violating input constraints. Finally, we demonstrated the algorithm on a linear system and showed that the new MDL string satisfied the input constraints, while still moving the state to our desired destination.

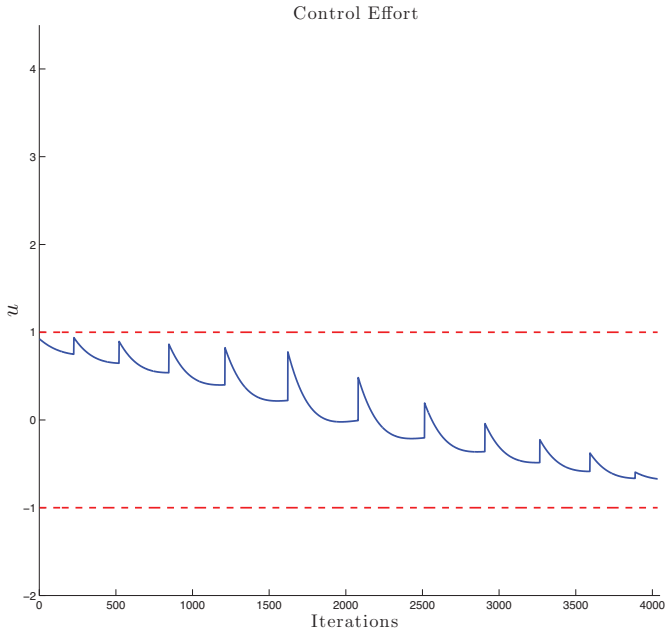


Figure 33: In this plot, we see the successful maintenance of the control bound $|u| < 1$ during the execution of the expanded MDL string.

CHAPTER V

ENABLING SOFTWARE FOR CYBER-PHYSICAL SYSTEMS

We now switch perspectives and consider the development of a software tool that can execute high-level control specifications for CPS. As discussed in [54], one key research area for cyber-physical systems (CPS) is the development of middleware systems that can "[satisfy] real-world physical constraints while providing multidimensional quality of service." As such, this chapter proposes and demonstrates a new middleware that enables the design of control software for CPS. This middleware, called *Pancakes*, gives CPS control application designers several key features. First, Pancakes provides a structured way to dynamically adjust the runtime behavior of components in the architecture according to changes in the environment. This dynamic adjustment allows for the construction of control applications that can respond to environmental and local changes to the CPS platform. An additional feature of Pancakes is its actor-oriented design (e.g. [2, 36]), which allows for the implementation of concurrent CPS control applications. Furthermore, Pancakes abstracts sensing, actuation, and networking capabilities so that high-level controllers can be implemented without worrying about low-level hardware management. Finally, Pancakes provides a logging service that facilitates more thorough evaluation of system operation; this feature helps CPS developers refine or revamp system tasks on future iterations of the control application.

Pancakes was inspired by the current literature in distributed and software control middleware, e.g. [1, 19, 38, 57], as well as robotics control software, e.g. [15, 27, 35,

50]. In [1], the authors developed a software framework that enabled the dynamic adjustment of a web server using feedback control. Their middleware exposed software "knobs" that could be tweaked to get better quality of service. Moving beyond this idea of modifying parameters of software components is the idea of reflective middleware [57]. This work describes a system where the pieces of the middleware dynamically adapt their capabilities as changes occur within the software. The work in [19] proposed a larger distributed embedded system framework that enables the development of software across many different types of computing platforms, from embedded controllers to desktop systems. Finally, the authors of [38] developed a system that allows for sensor fusion and provides other facilities, such as clock synchronization, to help implement distributed sensing algorithms on resource constrained platforms.

The work in robotics software architectures made the control of heterogeneous systems easier by abstracting the sensors and actuators. For example, [27] created a common interface to the sensors and actuators of the robots so that users could write control software that works on different types of robots without having to know every detail of the robot's implementation. The authors of [15] took this idea a step further by separating the capabilities of a robot into discrete, re-usable components that can be assembled into a larger robot control application. Additionally, the work in [35] applied multi-agent software design to create a platform for developing distributed robotics applications. These robotics software frameworks successfully demonstrate that abstracting control and sensing devices makes it easier to develop complex, distributed control applications.

The structure of this chapter is as follows: Section 5.1 describes the design of the Pancakes architecture and its operational components. Section 5.2 discusses the implementation of the current version of Pancakes as well as an example control application constructed using this framework. Following this discussion, we demonstrate Pancakes-based control applications executing on mobile robots and sensor nodes in

Section 5.3.

5.1 System Architecture

This section describes the main pieces of the Pancakes architecture and how they work together when creating CPS applications. Figure 34 illustrates the core of the pancakes architecture when deployed onto a system. The Pancakes kernel (shaded circles) sets up the infrastructure for interacting with the local system, as well as any networked systems. The kernel is composed of two pieces: the scheduler and the information stream. The scheduler accepts system tasks that require a periodic execution, such as a battery sensor or agent discovery. The information stream creates the channels through which all Pancakes components pass messages. Additionally, the kernel launches required and optional services that are provided in a configuration file. The diagram in Figure 34 shows the main services, Client, Network, Device, and Logging, as well as the optional Custom services that are created by the user.

5.1.1 Information Stream

To maintain a high-level of parallelism and limit the amount of thread blocking, Pancakes uses message-based communication among the system components. The information stream sets up the communication channels that services and tasks can publish new information, or subscribe to receive information from other Pancakes components. This stream contains five core channels: `system`, `control`, `network`, `log`, and `command`. Additionally, services can create specialized channels that are used to pass service specific information among tasks within the service. For example, a client service that spawns a battery monitoring task can create a new channel, `batterymon`, that other tasks can listen to for updates from the battery monitor.

The `system` channel provides a channel for system services and tasks to publish information to user-made and other system tasks. For example, a mobile robot's sonar sensor task would publish its most recent data points to the `system` channel,

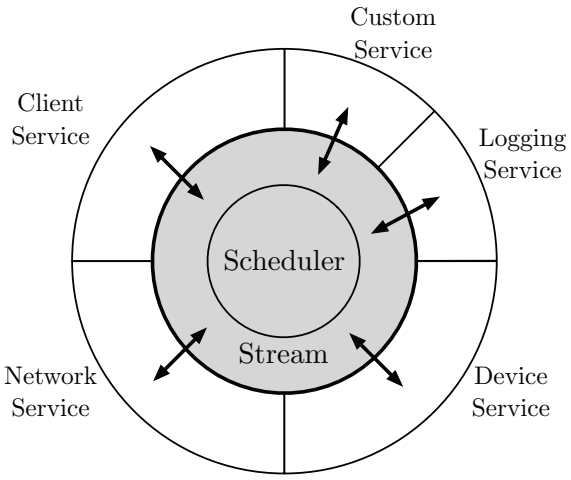


Figure 34: An illustration of the inner structure of Pancakes. The kernel (shaded circles) maintains the scheduler and main information stream that contains all system channels. System services are launched by the kernel when the agent is launched.

which is subscribed to by a user-made task. The `control` channel serves as a control messaging channel among the services and user tasks. Messages sent over this channel enable the dynamic rescheduling or shut-down of tasks and services.

The `network` channel handles the transmission of messages to and from Pancakes agents. A user-made task that requires network communication publishes its network messages to this channel, which the network service then routes to the desired Pancakes agent on the network. The `log` channel allows any Pancakes component to perform error, debug, or data logging, which helps in the post-run analysis and debugging of complex, distributed applications. Finally, to issue control commands to tasks that can actuate, such as a motor device, tasks send messages over the `command` channel.

5.1.2 System Services

The Pancakes system services, illustrated by the outer ring in Figure 34, spawn and maintain all desired capabilities for a CPS platform. In this section, we describe the functionality of each of these services.

The Device Service creates the system device tasks, which are the hardware abstractions for sensors and actuators, and schedules any tasks that require timed execution. Figure 35 shows an example configuration for a mobile robot's Device Service. In this case, the robot gets sensor information from the sonar array and battery device tasks, which publish their data to the `system` channel within the Pancakes information stream. The motor task listens to the `command` channel for requests to execute a motor command.

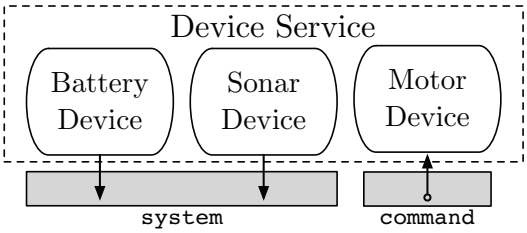


Figure 35: The device service spawns several tasks that provide information to all `system` channel subscribers, and enables the control of actuators, such as robot motors. The configuration shown in this figure has two devices that publish information, battery and sonar, and the motor device that accepts motor speed commands.

Another key service is the Network Service, which enables communication with other Pancakes agents on the network. This service creates a network client task that listens to the `network` channel for any outgoing network messages and transmits them to the intended target. To perform agent discovery, which dynamically builds up the network neighborhood of the agent, the Network Service launches the discovery speaker and listener tasks. The standard configuration for the Network Service uses TCP sockets for network connections. However, one could develop new tasks that implement advanced ad-hoc routing policies, which are increasingly important on

large, geographically sparse networks.

The Client Service spawns user tasks and channels establishing their internal communication. Users implement tasks to carry out communication and control algorithms, which are then loaded into the Client Service at system startup. The Logging Service listens to the `log` channel and displays error and debug messages to screen; additionally, it can record messages to a file for later analysis. Finally, if a user has implemented any Custom Services for the Pancakes architecture, they are enabled using a configuration file.

5.1.3 Tasks

Task components are the main "actors" in Pancakes: they produce and consume information in order to affect a change in the deployed system. Tasks can execute in time-driven, event-driven, or a combination of both modes, depending on the desired functionality set by the designer. Since these tasks can only communicate via the Pancakes stream channels, there is no need to synchronize on shared variables. Instead, variables are transmitted through the channels to subscribing Pancakes components.

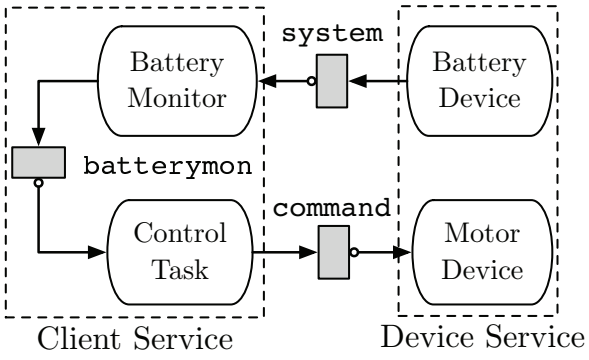


Figure 36: This figure illustrates an example application where the Client Service monitors the battery voltage to modify the maximum driving speed of the mobile robot. Note that the Client Service creates a new `batterymon` channel for the Battery Monitor to publish information to the Control task.

Figure 36 illustrates a Pancakes application that modifies a mobile robot's top

speed parameter based on the robot's current battery voltage. The BatteryMonitor task receives raw data from the BatteryDevice (a timed task) and processes it to determine if a particular threshold is reached. It then publishes a message on the `batterymon` channel, to which the ControlTask subscribes. The ControlTask receives the message, alters its local max speed variable, and then continues to publish motor commands onto the `command` channel. Without the messaging capability, these individual threads would need to synchronize on variables, such as the battery level or maximum speed values, which could potentially cause a bottleneck during execution.

5.1.4 Dynamic Adjustment of Components

One of the key features of Pancakes is the ability to adjust services and tasks as an application executes on a target system. This feature lets the application adjust the capabilities within the architecture according to dynamic effects from software (i.e. logic statements, software controllers) or the physical environment (i.e. power consumption, sensing data). For example, a task can request that network discovery be slowed down to reduce the number of network transmissions, and, hence, reduce the rate of power consumption of the application. Also, a more drastic power savings could be achieved by requesting the Network Service to shut down temporarily.

We enable this feature by establishing a messaging protocol for tasks and services to request changing other task and service runtime behavior. The currently supported control operations are *cancel*, *restart*, or *reschedule*. For a task or service to initiate one of these controls, it must send a control message over the `control` channel within Pancakes. All services subscribe to this channel, and each looks at the message to determine if it is the service to change, or if it has a task that must be cancelled or rescheduled.

Once the message is received at the target service, the service calls on the kernel to cancel or reschedule the task. Additionally, if the service is requested to stop

or restart, it shuts down all of its currently running tasks and requests the kernel to stop/restart itself.[1] In the next section we discuss the implementation details of Pancakes and describe how a Pancakes application is developed.

5.2 Control Application Development

Pancakes is implemented with Java to ensure that it can operate on several types of computational platforms and operating systems. Another reason we use Java to implement Pancakes is the existence of robust Java libraries that enable concurrency and message passing. The Java SE 6^{TM}standard library has new concurrency tools that efficiently handle multiple threads using specialized thread pools. Complementary to this library, we make use of the Jetlang[2] library, which is a Java package that provides messaging services for multi-threaded applications.

Using these libraries, we developed the Pancakes architecture to be easily extended by enabling the creation of services and tasks based on mission goals for a particular CPS application. Since these services and tasks all communicate with internal messages, not shared access to data, application design focuses on the input/output behavior of each task. First, we decide what behavior a task needs to perform and determine the information necessary to execute the behavior. Then, we create the Pancakes system with the required devices (sensors and actuators) and networking capabilities. An example of this process is described in the next section, where we create a multi-agent Pancakes application to be used in our experiments in Section 5.3.

[1] Note that in its current state, Pancakes allows control requests to be processed without checking if a task has the *permission* to do so; however, future revisions will incorporate security policies to prevent malicious controls from changing the system operation.

[2] *http://code.google.com/p/jetlang/*

5.2.1 Example Control Application

Consider the team of mobile robots and wireless sensors shown in Figure 37, which are deployed in a building for security monitoring. The sensor nodes establish a perimeter to detect an intruder and transmit this information to the mobile robots. The team of robots explore the area, sharing local information among themselves, such as senor data or power levels, to make a coordinated decision for interception of the intruder. Once an intruder is detected, as shown in Figure 37(a), the sensor nodes inform the robots, which each change their mode of operation to encircle the intruder. Figure 37(b) shows the ideal case where all three robots are able to encircle the intruder.

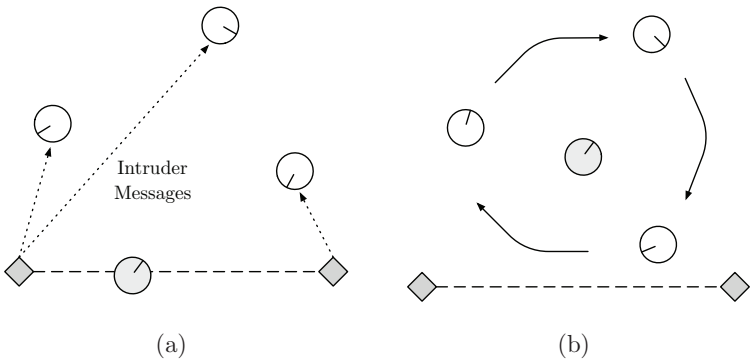


Figure 37: An illustration of an example scenario requiring a team of robots (white circles) and sensor nodes (grey diamonds) to work together to surround an intruding robot (grey circle).

One key challenge with this set of systems is that they must be aware of their communication and power capabilities when making control decisions. If a sensor node has been too aggressive with its messaging, it may not have enough energy to transmit sensor information when an intrusion has occurred. Alternatively, if a robot is wandering aimlessly, its actuators will drain its battery and it will not be able to initiate a capture of the intruder.

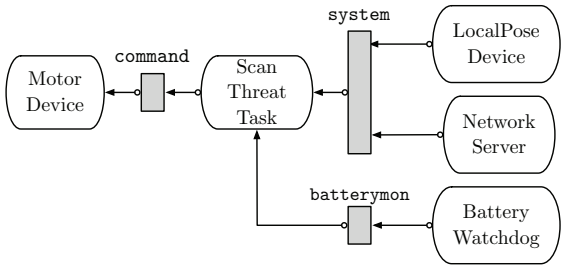
Instead we want to construct control applications for these systems that take into account their physical capabilities, such as power, and environmental changes faced

during mission execution. Using Pancakes to develop the control applications, we create an adaptive communication task for the sensor nodes that adjusts how often it sends messages when no intruder is detected. Furthermore, we can design the robots' control applications to monitor the battery levels and reduce their inter-agent sensor updates until they receive an "intruder" message from the sensor nodes.

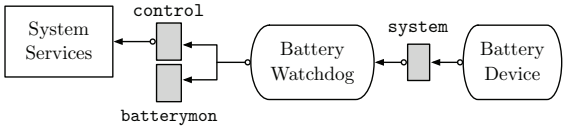
Here we develop and test one of the controllers described in this scenario: the robot encirclement behavior. We implement a dynamic boundary tracking controller, first formulated in [63], to serve as our encircling behavior to contain an intruder. Another design goal of this control application is to conserve power on the robots once they have consumed battery beyond some threshold value. By conserving power at runtime, the robots could remain deployed for a longer period of time. To achieve this goal, we create tasks that adjust the control parameters and tasks that monitor the dynamic battery levels of the robots. A full diagram of the control application is shown in Figure 38.

For this example, we create a client service that spawns three tasks: ScanThreat, which calculates the boundary tracking algorithm, BatteryWatchdog, which monitors the battery levels of the system, and LocalPose Sharing, which transmits the agent's local pose data to other Pancakes agents. Hence, we need the ScanThreat task, illustrated in Figure 38(a), to listen for its local pose information *and* the local pose of its neighbors, which are delivered over the `system` channel by the LocalPose device and Network Server, respectively. Also, ScanThreat must change its control parameters when the battery drops below a particular level, so it listens to the client channel `batterymon`. The ScanThreat task is event driven, so it only executes its algorithm when local pose information is received on its input.

The BatteryWatchdog in Figure 38(b) listens to the `system` channel for battery value updates, and transmits messages over `batterymon` when the battery level has crossed certain thresholds. Additionally, this task sends requests over `control` to



(a) ScanThreat Task



(b) BatteryWatchdog Task

(c) LocalPose Sharing Task

Figure 38: These figures illustrate the three main client tasks and their information dependencies. ScanThreat, LocalPose Sharing, and Battery Watchdog, subscribe to channels that deliver the appropriate information for their execution.

alter the LocalPose Device and LocalPose Sharing update frequencies in order to conserve power. These control messages are received by the system services, which all subscribe to `control`. In this application, the Device Service will reschedule the LocalPose Device at a slower update rate to reduce the number of local pose updates that are sent to the ScanThreat task. Additionally, the Client Service will reschedule the LocalPose Sharing task to transmit network updates at a slower frequency to the agent's neighbors.

The LocalPose Sharing task (Figure 38(c)) sends the agent's local pose data to neighbor agents by creating messages that are sent over the `network` channel. This task is both event and time driven, since it stores new local pose data when it receives new data from `system` and it runs at a set frequency to transmit the currently cached value to its neighbors. This task uses a handshaking protocol with the other agents

to prevent them from being "spammed" with too much information. The devices and network related tasks are all maintained by the Device and Network Services. The execution of this application on our experimental platform is described in the next section.

5.3 Experimental Results

In this section we demonstrate the Pancakes application described in Section 5.2.1. The platforms used in our experiment are shown in Figure 39. The application runs on Khepera III robots[3] (Figure 39(a)), which perform the target encirclement behavior described in the example scenario of Section 5.2.1. To determine the local pose of each robot we use a Vicon^{TM}motion capture system (Figure 39(b)). Since this local pose data is produced off-board by the motion capture system, it is a "virtual" local sensor on each robot. The Vicon^{TM}system tracks the reflective points on each robot and transmits the local pose data to each robot, where the data is received and handled by a LocalPose device in Pancakes. Additionally, we use BUGLabs[4] embedded computers as wireless sensor nodes that monitor the room for motion.

Initially, we test a single robot to show in detail how the Pancakes system enables the power-aware control of the robot. Following this initial experiment we deploy a second robot that executes the same application. However, since there is another agent on the network, the robots exchange information to tweak their speed calculations using a spacing algorithm. In the final experiment, we deploy a robot and BUG sensor node to imitate the intrusion detection scenario described in the prior section.

5.3.1 Results: Single Robot

In the single robot case, we set the origin of the environment as the target point to encircle. The robot continually executes the ScanThreat task, which has two

[3] *http://www.k-team.com*
[4] *http://www.buglabs.net/*

(a) Khepera III robots. (b) Vicon^{TM}motion capture system. (c) BUGLabs sensor node.

Figure 39: This figure shows the hardware devices used in our experiment. 39(a) shows the two K-Team robots that used Pancakes. These robots were provided with indoor localization data from the motion capture systems shown in 39(b). Furthermore, a BUGLabs embedded computer serves as a wireless sensor node in the testbed.

operational modes: high battery level and medium battery level. When operating with high battery level, the robot drives with a maximum speed of 0.13 meters/second and receives local pose updates 4 times a second. Once the battery dips below a set voltage threshold we reduce the speed to 0.06 meters/second and the local pose update rate to 2 times a second.

Figure 40 shows the logged motor commands issued by the ScanThreat task during our experiment. Up until the switch point, the robot tracked a large circular boundary, shown in Figure 41. Unfortunately, during execution, noise crept into the motion capture system and caused the local pose data to be corrupted. These errors resulted in the robot moving beyond the desired circular orbit (dotted trajectory line).

Once the battery dropped below our specified threshold, the Battery Monitor task sends messages to the ScanThreat task and the Device Service. ScanThreat changes its internal control variables to lower its maximum speed and adjust the tracking controller's gain. At the same time, the Device Service takes the control message from the Battery Monitor to adjust the LocalPose update rate to only 2 updates a second. The consequence of these changes is shown in both Figure 40 and 41, where

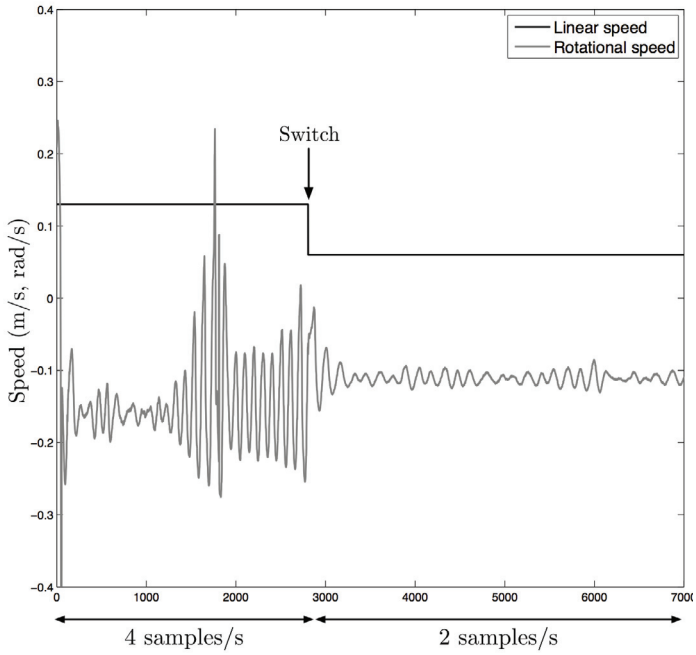


Figure 40: This figure shows the motor commands our controller issued to the robot during the execution of this application. While the robot receives the noisy "GPS" data, the algorithm over-corrects too much, resulting in higher rotational speeds.

the robot moves more slowly and tracks a smaller circle. After several seconds, the robot converges to the smaller sized circle (gray trajectory line) successfully.

In addition to correctly executing our desired behavior and modifying the run-time capabilities of the system, this experiment shows that Pancakes can be used to monitor and change power consumption in CPS applications. Figure 42 shows the power consumption of the robot as it executed the application. During the high battery level mode, the robot consumes, on average, to a little less than 4.4W. Once the agent switches, the average power consumption drops to about 4.25W.

5.3.2 Results: Robot Team

We expand our initial experiment by introducing a second robot into the environment. These robots execute the same application used in Section 5.3.1, but the robots are now able to share information using their LocalPose Sharing tasks.

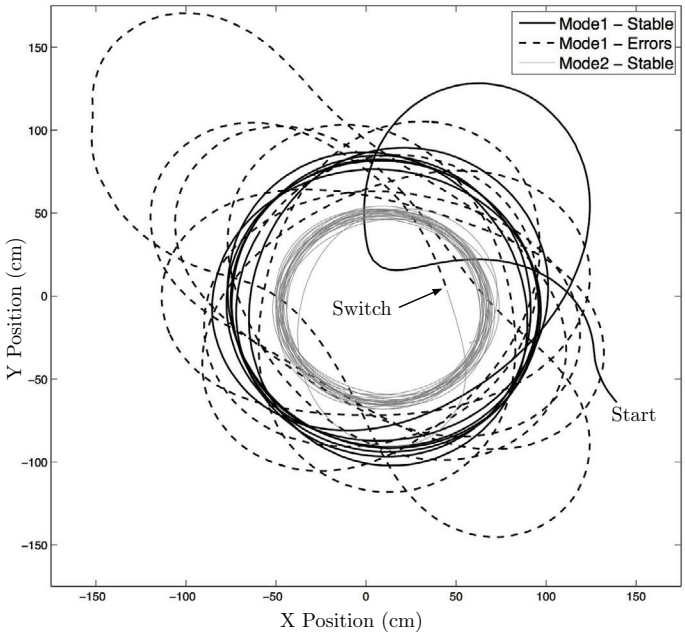


Figure 41: This figure shows the measured position delivered to the ScanThreat task from the LocalPose device. At first the robot converges to the circular boundary well, but after some noise entered our motion capture system, the controller over-compensates. Once the robot switches modes, it tracks the smaller boundary without as much error.

In the first run, the robots execute their boundary tracking behavior around the target, and, as Figure 43 shows, the agents converge to the large circular boundary. Additionally, they execute the spacing controller embedded within ScanThreat to keep themselves from colliding while circling the target. However, in this run, the battery measurement device onboard the Khepera III robots failed, which prevented the BatteryWatchdog from rescheduling and modifying the control algorithm.

The second run of our experiment is shown in Figure 44. This experiment shows the two robots starting to track their initial large circles; however, Agent 2's battery level drops below the threshold value, which switches its controller to track the smaller circle at slower speeds and sensor update rates. Additionally, the robot is still attempting to perform its spacing calculation with Agent 1, which is on the

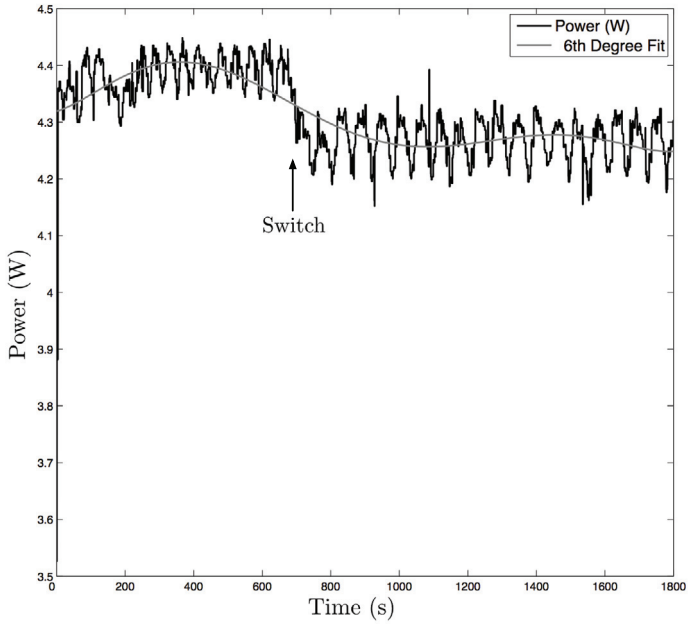


Figure 42: The graph in this figure shows the power consumption calculated from the voltage and current data on the robot. Note that after our switch to the lower power mode (slower motors, slower update rates) the robot indeed reduces its power consumption rate.

boundary of the outer circle. The slower, internal local pose update rates and the fact that Agent 2 is tracking a different boundary from Agent 1 results in a "clover leaf" motion around the origin.

These robot team experiments demonstrate that Pancakes facilitates the development of power-aware, multi-agent control software. Additionally, Pancakes gives us the ability to extract many types of runtime data, which imparts further insight into the design of our multi-agent application as well as the design of our experimental apparatus.

5.3.3 Results: Robot and Sensor Node

This experiment integrates a sensor node with one of our robots to demonstrate Pancakes' ability to deploy applications on a heterogeneous network. In this scenario, we implement a motion detection task on the BUG that is inspired by our example

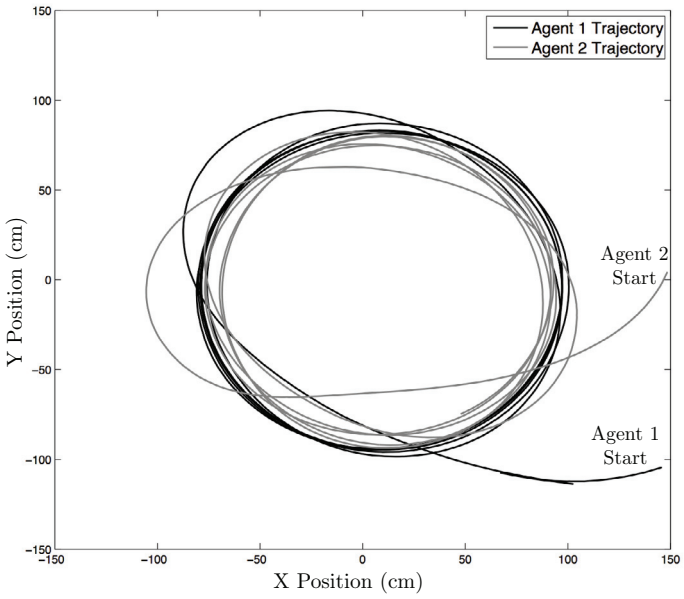


Figure 43: This figure shows the trajectories of the two robots as they track a circular path around the origin. Each one shares information using their LocalPose Sharing tasks, which allows them to maintain spacing and not overtake each other during execution.

sensor nodes in Section 5.2.1. This task monitors the information from the motion sensor device on the BUG and transmits the nodes location whenever it detects motion nearby.

Figure 45 shows the trajectory of the robot and the BUG's position during our experiment. The robot idles at its start position until the BUG transmits its location data when someone walks nearby. The robot immediately sets the location as its current target and begins the `ScanThreat` behavior. After encirclement of the target is achieved, we move the BUG to another location in the room and trigger its motion sensor. Again, the robot sets its target location and approaches the new threat.

5.3.4 Experimental Evaluation

Overall, our experimental results demonstrate the challenges of CPS control application development. The successful experimental runs show that Pancakes lets us

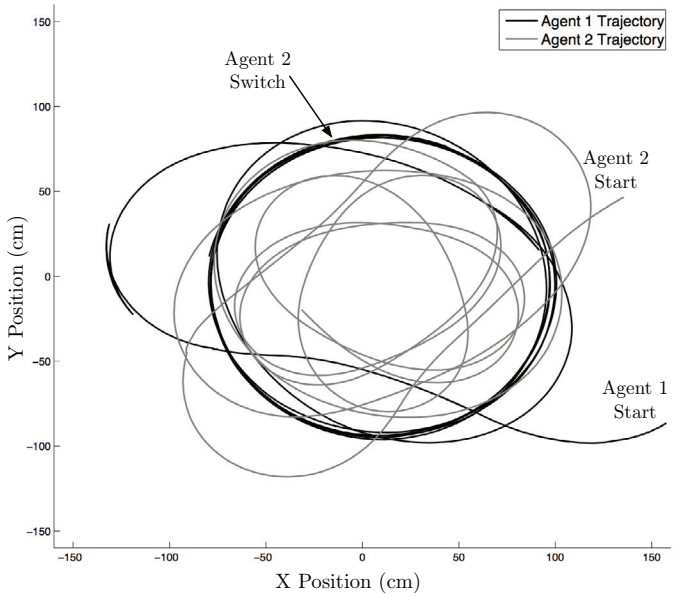


Figure 44: The second run of our two robot experiment shows the robots initially tracing the boundary. However, Agent 2's power level dips below the threshold and it is rescheduled to run at a slower rate and speed. At the same time, the spacing algorithm introduces errors on the speed control, resulting in the "clover leaf" shape shown in gray.

implement power-aware, multi-agent controllers that can make runtime adjustments to the architecture's components and control parameters to achieve a design goal, i.e. reduce power consumption. The experimental results with runtime errors demonstrated that our control application is not robust due to errors in the sensors or limited communication bandwidth. These issues are runtime constraints that are not easily identifiable before the deployment. Pancakes helps us identify such constraints by logging all information during the experiment. We can use this information to implement more robust control applications while addressing the physical constraints seen in deployment. Therefore, Pancakes is a tool to implement better control applications for CPS platforms. Future revisions of Pancakes will also work towards providing more architectural features, such as real-time support and security protocols to prevent malicious adjustment of the architecture.

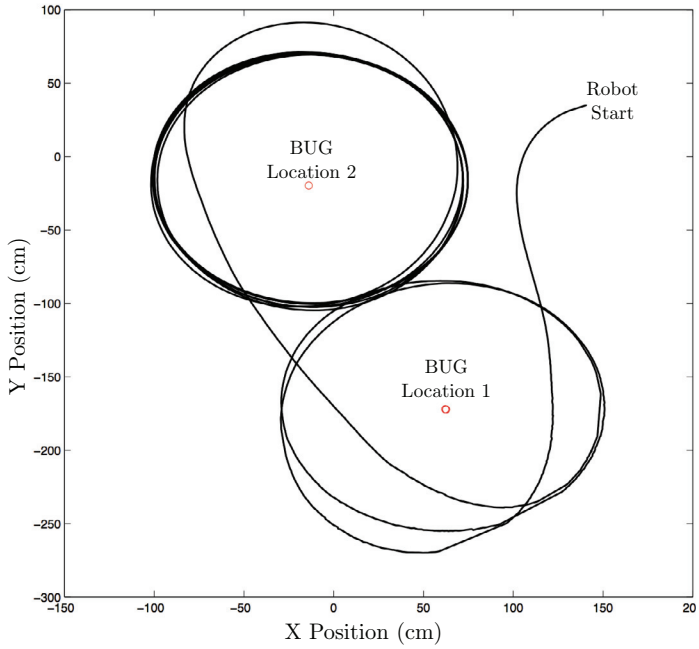


Figure 45: This figure shows the trajectory of the robot as it tracks the location of the BUG sensor node. We moved the node one time, which then caused the robot to switch targets based on incoming messages from the BUG agent.

5.4 Conclusions

This chapter presented the development of a new software architecture that facilitates the development of concurrent, agent-based control applications for cyber-physical system applications. We described the architecture design and implementation as well as gave an example of a Pancakes-based control application involving mobile robots. Furthermore, we demonstrated the successful deployment and execution of the example control application onto a team of mobile robots and wireless sensor nodes.

CHAPTER VI

CONCLUSIONS AND FUTURE DIRECTIONS

6.1 Conclusions

In this thesis, we developed design and optimization tools for the generation of executable control code from high-level, symbolic specifications. This effort works toward bridging the gap between the specification of control programs and their execution on the target systems. The main contributions of this thesis are:

- *Spatio-temporal motion program optimization.* We derived optimality conditions for an optimal control problem involving switched-systems with spatial, temporal, and energy constraints. Furthermore, these optimality conditions serve as the basis for a numerical algorithm that "compiles" motion programs for such systems and outputs optimized control code based on new temporal and energy parameters. This work was demonstrated on simulated and robotic marionettes.

- *A motion program framework for networked systems.* We developed a new motion description language, MDLn, that encodes the desired communication topology of an agent. Using MDLn allows us to specify control for a team of networked systems whose controllers and interrupts require information from these preferred neighbors, or buddies. We developed an algorithm to automatically generate a supervisor that prevents the systems from executing incorrect MDLn programs. This work was demonstrated on simulated unmanned vehicles.

- *Expansion of motion programs to maintain actuator bounds.* We developed a method to determine if an MDL specification can execute on a system under

actuator constraints. Furthermore, if this specification does not lead to correct execution, an we use an algorithm to insert new MDL modes that maintain the actuator bounds for the length of the motion program. This algorithm was demonstrated in simulation.

- *A software framework for controlling cyber-physical systems.* We designed and implemented a new software architecture that facilitates the development of control software for cyber-physical systems. This architecture provides important features for heterogeneous, networked systems requiring control software that is physically-aware, i.e. can handle power, communication, actuator, and sensing constraints. This architecture was demonstrated on Khepera III robot platforms and a BUG Labs sensor node. Additionally, this software served as the underlying enabling software for the implementation of the MDLn framework.

These technical contributions are supported by the following publications: [41, 42, 43, 44, 45, 47, 46].

6.2 Future Directions

The work in this thesis presents a starting point for further exploration of bridging the specification-to-execution gap, as described in Chapter 1. To advance the ideas presented in this thesis, more work remains in developing new computational algorithms that incorporate more physical and computational constraints. Furthermore, the enabling software architecture for CPS platforms, needs more research in real-time, distributed computing, which is a key requirement of future CPS applications.

The spatio-temporal motion program framework of Chapter 2 encoded spatial constraints as an element of the cost functional in the optimization process. Although, this cost allows for a reasonable correction of spatial specifications with MDLp programs, there are no hard guarantees that the agent would wind up in the region.

Future applications that have real spatial boundaries require that the MDLp compiler deal with hard constraints.

The MDLn framework in Chapter 3 did succeed in defining and executing motion programs for collections of agents. However, the buddy lists and roles of the agents are all specified as *static* entities. In real world applications, this situation may not always be the case. For example, an agent may depend on information from the "closest" agent, rather than a pre-determined agent. The roles of agents may also need to be dynamic depending on what happens to the agents as they execute their motion programs. Further development of these ideas would strengthen the MDLn framework since it would be able to handle more diverse multi-agent applications.

Chapter 4 developed of an algorithm to identify and correct a specified MDL string that operates under bounded input. However, the algorithm is limited to single input, linear systems. Since CPS platforms will more than likely have multiple inputs, future research in this area will require generalization of the problem to more complicated systems with multiple inputs.

Finally, the Pancakes architecture in Chapter 5 facilitates the design and implementation of control applications for CPS that can react to the current power, actuator, sensing, and communication capabilities of the system. This software requires more work to push it into the real-time realm since most CPS platforms will be based on small, embedded systems with real-time computation and communication constraints. Finally, to assist in bridging the specification-to-execution gap, Pancakes would benefit from having its own high-level control abstraction that could be inserted into motion programs. This ability would be a first step towards connecting the control specifications with the hardware capabilities of the target system (or systems).

REFERENCES

[1] Abdelzaher, T., Stankovic, J., Lu, C., Zhang, R., and Lu, Y., "Feedback performance control in software services," *IEEE Control Systems Magazine*, pp. 74–90, June 2003.

[2] Agha, G., "Concurrent object-oriented programming," *Communications of the ACM*, vol. 33, pp. 125–140, September 1990.

[3] Armijo, L., "Minimization of functions having lipschitz continuous first-partial derivatives," *Pacific Journal of Mathematics*, vol. 16, pp. 1–3, 1966.

[4] Arrow, K., Hurwicz, L., and Uzawa, H., *Studies in Nonlinear Programming*. Stanford University Press, 1958.

[5] Attia, S., Alamir, M., and de Wit, C. C., "Sub optimal control of switched nonlinear systems under location and switching constraints," in *Proceedings 16th IFAC World Congress*, July 2005.

[6] Baird, B., *The Art of the Puppet*. New York: McMillan Company, 1965.

[7] Belta, C., Bicchi, A., Egerstedt, M., Frazzoli, E., Klavins, E., and Pappas, G., "Sybolic planning and control of robot motion," *IEEE Robotics and Automation Magazine*, March 2007.

[8] Bicchi, A., Marigo, A., and Piccoli, B., "Encoding steering control with symbols," in *Proceedings of 42nd IEEE Conference on Decision and Control*, pp. 3343–3348, 2003.

[9] Bicchi, A., Marigo, A., and Piccoli, B., "Feedback encoding for efficient symbolic control of dynamical systems," *IEEE Transactions on Automatic Control*, vol. 51, pp. 987–1002, June 2006.

[10] Blanchinin, F., Pellegrino, F. A., and Visentini, L., "Control of manipulators in a constrained workspace by means of linked invariant sets," *Intenational Journal of Robust and Nonlinear Control*, vol. 14, pp. 1185–1205, 2004.

[11] Branicky, M., *Studies in Hybrid Systems: Modeling, Analysis, and Control*. PhD thesis, Massachusetts Institute of Technology, 1995.

[12] Branicky, M., "Introduction to hybrid systems," in *Handbook of Networked and Embedded Control Systems* (Hristu-Varsakelis, D. and Levine, W., eds.), pp. 91–116, Boston: Birkhäuser, 2005.

[13] Brockett, R., "On the computer control of movement," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, pp. 534–540, 1988.

[14] Brockett, R., "Hybrid models for motion control systems," in *Essays in Control* (Trentelman, H. and Willems, J., eds.), pp. 29–53, Boston: Birkhäuser, 1993.

[15] Brooks, A., Kaupp, T., Makarenko, A., Williams, S., and Oreback, A., "Towards component-based robotics," in *Proceedings of the International Conference on Intelligent Robots and Systems*, pp. 163–168, 2005.

[16] Burridge, R., Rizzi, A., and Koditschek, D., "Sequential composition of dynamically dexterous robot behaviors," *International Journal of Robotics Research*, vol. 8, pp. 534–555, June 1999.

[17] Cassandras, C. and Lafortune, S., *Introduction to Discrete Event Systems*. Norwell, MA: Kluwer Academic Publishers, 1999.

[18] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., *Introduction to Algorithms*. Cambridge, Massachusettes: MIT Press, 2001.

[19] Cornea, R., Dutt, N., Gupta, R., Krueger, I., Nicolau, A., Schmidt, D., and Shukla, S., "Forge: A framework for optimization of distributed embedded systems software," in *Proceedings of the 17th IEEE/ACM International Parallel and Distributed Processing Symposium*, 2003.

[20] Delmotte, F. and Egerstedt, M., "Reconstruction of low-complexity control programs from data," in *Proceedings of 43rd IEEE Conference on Decision and Control*, vol. 2, pp. 1460–1465, December 2004.

[21] Egerstedt, M., "Motion description languages for multi-modal control in robotics," in *Control Problems in Robotics, Springer Tracts in Advanced Robotics* (Bicchi, A., Cristensen, H., and Prattichizzo, D., eds.), pp. 75–90, Springer-Verlag, December 2002.

[22] Egerstedt, M. and Brockett, R., "Feedback can reduce the specification complexity of motor programs," *IEEE Transactions on Automatic Control*, vol. 48, pp. 213–223, February 2003.

[23] Egerstedt, M., Murphey, T., and Ludwig, J., "Motion programs for puppet choreography and control," in *Hybrid Systems: Computation and Control*, (Pisa, Italy), pp. 190–202, Springer-Verlag, April 2007.

[24] Egerstedt, M., Wardi, Y., and Axelsson, H., "Trasition-time optimization for switched-mode dynamical systems," *IEEE Transactions on Automatic Control*, vol. 51, pp. 110–115, January 2006.

[25] Frazzoli, E., Dahleh, M., and Feron, E., "A hybrid control architecture for agressive maneuvering of autonomous helicopters," in *Proceedings of 38th IEEE Conference on Decision and Control*, pp. 2471–2476, December 1999.

[26] Frazzoli, E., Dahleh, M., and Feron, E., "Maneuver-based motion planning for nonlinear systems with symmetries," *IEEE Transactions on Robotics*, vol. 21, pp. 1077–1091, December 2005.

[27] Gerkey, B., Vaughan, R. T., and Howard, A., "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th Annual International Conference on Advanced Robotics*, pp. 317–323, June 2003.

[28] Goebel, R., Sanfelice, R., and Teel, A., "Hybrid dynamical systems," *IEEE Control Systems Magazine*, vol. 29, pp. 28–93, April 2009.

[29] Hopcroft, J., Motwani, R., and Ullman, J., *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[30] Hristu-Varsakelis, D., Egerstedt, M., and Krishnaprasad, P., "On the structural complexity of the motion description language mdle," in *Proceedings of 42nd IEEE Conference on Decision and Control*, 2003.

[31] Johnson, E. and Murphey, T., "Dynamic modeling and motion planning for marionettes: rigid bodies articulated by massless strings," in *IEEE International Conference on Robotics and Automation*, pp. 330–335, April 2007.

[32] Kloetzer, M., *Symbolic Motion Planning and Control*. PhD thesis, Boston University, 2008.

[33] Kloetzer, M. and Belta, C., "Temporal logic planning and control of robotic swarms by hierarchical abstractions," *IEEE Transactions on Robotics*, vol. 23, pp. 320–330, April 2007.

[34] Kolmanovsky, I. and Gilbert, E., *Mulitmode Regulators for Systems with State and Control Constraints and Disturbance Inputs*. Lecture Notes in Control and Information Sciences, Springer-Verlag, 1997.

[35] Kulis, Z., Manikonda, V., Azimi-Sadjadi, B., and Ranjan, P., "The distributed control framework: A software infrastructure for agent-based distributed control and robotics," in *Proceedings of American Control Conference*, 2008.

[36] Lee, E. A., "Model-driven development - from object-oriented design to actor-oriented design," in *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*, (Chicago), 2003.

[37] Luenberger, D., *Optimization by Vector Space Methods*. John Wiley and Sons, Inc., 1969.

[38] Mangharam, R., Rowe, A., and Rajkumar, R., "Firefly: A cross-layer platform for wireless sensor networks," *Real Time Systems Journal*, November 2006.

[39] Manikonda, V., Krishnaprasad, P. S., and Hendler, J., "Languages, behaviors, hybrid architectures and motion control," in *Mathematical Control Theory* (Willems, J. and Baillieul, J., eds.), Springer-Verlag, 1998.

[40] Manikonda, V., Krishnaprasad, P., and Hendler, J., "A motion description language and a hybrid architecture for motion planning with nonholonomic robots," in *IEEE International Conference on Robotics and Automation*, vol. 2, pp. 2021–2028, May 1995.

[41] Martin, P., de la Croix, J., and Egerstedt, M., "MDLn: A motion description language for networked systems," in *Proceedings of 47th IEEE Conference on Decision and Control*, 2008.

[42] Martin, P. and Egerstedt, M., "Optimal timing control of interconnected, switched systems with applications to robotic marionettes," in *Workshop on Discrete Event Systems*, (Gothenburg, Sweden), May 2008.

[43] Martin, P. and Egerstedt, M., "Motion description language-based topological maps for robot navigation," *Communications in Information and Systems*, vol. 8, no. 2, pp. 171–184, 2008.

[44] Martin, P. and Egerstedt, M., *Optimization of Multi-Agent Motion Programs with Applications to Robotic Marionettes*. Hybrid Systems: Computation and Control, Springer-Verlag, April 2009.

[45] Martin, P. and Egerstedt, M., "Timing control of switched systems with applications to robotic marionettes," *Journal of Discrete Event Dynamic Systems: Theory and Applications*, 2009.

[46] Martin, P. and Egerstedt, M., "Expanding motion programs under input constraints," in *American Control Conference*, (Baltimore, Maryland), June 2010.

[47] Martin, P. and Egerstedt, M., "On the specification and execution of motion programs for networked systems," in *Proceedings of the 19th International Symposium on Mathematical Theory of Networks and Systems*, 2010.

[48] McConley, M., Appleby, B., Dahleh, M. A., and Feron, E., "A computationally efficient lyapunov-based scheduling procedure for control of nonlinear systems with stability guarantees," *IEEE Transactions on Automatic Control*, vol. 45, pp. 33–47, January 2000.

[49] Moeseke, P. and de Ghellinck, G., "Decentralization in separable programming," *Econometrica*, vol. 37, no. 1, pp. 73–78, 1969.

[50] Montemerlo, M., Roy, N., and Thrun, S., "Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (carmen) toolkit," in *Proceedings of the International Conference on Intelligent Robots and Systems*, pp. 2436–2441, October 2003.

[51] Ramadge, P. and Wonham, W. M., "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, January 1989.

[52] Rantzer, A., "On price mechnanisms in linear quadratic team theory," in *Proceedings of 46th IEEE Conference on Decision and Control*, December 2007.

[53] Shaikh, M. and Caines, P., "On trajectory optimization for hybrid systems: Theory and algorithms for fixed schedules," in *Proceedings of 41st IEEE Conference on Decision and Control*, 2002.

[54] Stankovic, J., Lee, I., Mok, A., and Rajkumar, R., "Opportunities and obligations for physical computing systems," *Computer*, vol. 38, pp. 23–31, November 2005.

[55] Sussman, H., "Set-valued differentials and the hybrid maximum principle," in *Proceedings of 39th IEEE Conference on Decision and Control*, pp. 558–563, 2000.

[56] Tabuada, P. and Pappas, G., "Linear time logic control of discrete-time linear systems," *IEEE Transactions on Automatic Control*, vol. 51, pp. 1862–1877, December 2006.

[57] Wang, N., Kircher, M., and Schmidt, D., "Applying reflective middleware techniques to optimize a qos-enabled corba component model implementation," in *Proceedings of 24th Annual International Computer Software and Applications Conference*, pp. 492–499, October 2000.

[58] Witsenhausen, H. S., "A class of hybrid-state continuous-time dynamic systems," *IEEE Transactions on Automatic Control*, vol. 11, no. 2, pp. 161–167, 1966.

[59] Wredenhagen, G. and Belanger, P., "Piecewise-linear lq control for systems with input constraints," *Automatica*, vol. 30, no. 3, pp. 403–416, 1994.

[60] Xu, X. and Antsaklis, P., "Optimal control of switched autonomous systems," in *Proceedings of 41st IEEE Conference on Decision and Control*, (Las Vegas, Nevada), pp. 4401–4406, December 2002.

[61] Xu, X. and Antsaklis, P., "Optimal control of switched systems via nonlinear optimization based on direct differentiations of value functions," *International Journal of Control*, vol. 75, pp. 1406–1426, 2002.

[62] Yamane, K., Hodgins, J., and Brown, H., "Controlling a marionette with human motion capture data," in *IEEE International Conference on Robotics and Automation*, vol. 3, pp. 3834–3841, September 2003.

[63] Zhang, F., Justh, E., and Krishnaprasad, P., "Boundary following using gyroscopic control," in *Proceedings of the 43rd IEEE Conference on Decision and Control*, December 2004.

[64] Zhang, F., Goldgeier, M., and Krishnaprasad, P. S., "Control of small formations using shape coordinates," in *Proceedings of the International Conference on Robotics and Automation*, pp. 2510–2515, 2003.

[65] Zhang, W. and Tanner, H. G., "Composition of motion description languages," in *Hybrid Systems: Computation and Control* (Egerstedt, M. and Mishra, B., eds.), Springer-Verlag, 2008.